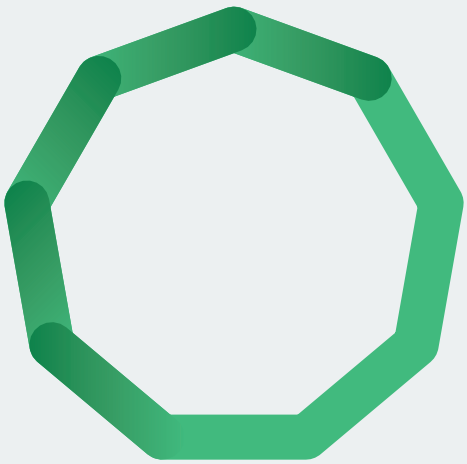


Distributed Systems

the source of truth

virtuoso edition

Feel free to contribute at `https://github.com/alexprut/distributed-systems`

First release, August 2018, version 1.0.0

Contents

Introduction **5**

1 Time, Clocks, and the Ordering of Events in a Distributed System **7**

2 Distributed Snapshots: Determining Global States of Distributed Systems **17**

3 Impossibility of Distributed Consensus with One Faulty Process **31**

4 The Byzantine Generals Problem **41**

5 The Part-Time Parliament **63**

6 Paxos Made Simple **97**

7 Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web **113**

8 Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications **127**

9 Bigtable: A Distributed Storage System for Structured Data **141**

10 Dynamo: Amazon's Highly Available Key-value Store **157**

11 The Chubby lock service for loosely-coupled distributed systems **175**

12 The Google File System **193**

13 MapReduce: Simplified Data Processing on Large Clusters **209**

14 Cassandra - A Decentralized Structured Storage System **223**

15 Paxos Made Live - An Engineering Perspective **231**

16 Spanner: Google's Globally-Distributed Database **249**

17 Spanner, TrueTime and the CAP Theorem **265**

18 ZooKeeper: Wait-free coordination for Internet-scale systems **273**

19 Kafka: a Distributed Messaging System for Log Processing **289**

20 A simple totally ordered broadcast protocol **297**

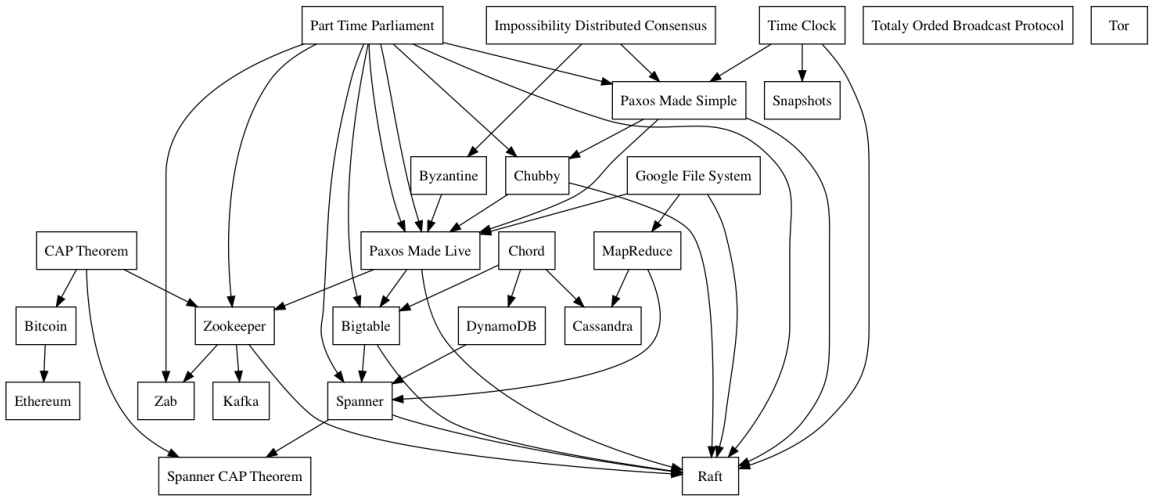
21 In Search of an Understandable Consensus Algorithm **305**

22 Bitcoin: A Peer-to-Peer Electronic Cash System **325**

23 Ethereum: a secure decentralized generalized transaction ledger **335**

Introduction

Both in the libraries and on the internet plenty of resources are available regarding the **Distributed Systems** topic. All of them are just a better explanation of the same exact source of truth, i.e. academic papers published by the brightest minds in the world. This book is just an ordered collection of all those publications. Below is depicted the order in which you can read the papers. The primary order is based on the citations dependency in the publications, secondary order is based on the year of publication and finally the last criteria is based on the topic dependency.



Chapter 1

Time, Clocks, and the Ordering of Events in a Distributed System

Authors:
Leslie Lamport

Publication year: 1978

Operating Systems

R. Stockton Gaines
Editor

Time, Clocks, and the Ordering of Events in a Distributed System

Leslie Lamport
Massachusetts Computer Associates, Inc.

The concept of one event happening before another in a distributed system is examined, and is shown to define a partial ordering of the events. A distributed algorithm is given for synchronizing a system of logical clocks which can be used to totally order the events. The use of the total ordering is illustrated with a method for solving synchronization problems. The algorithm is then specialized for synchronizing physical clocks, and a bound is derived on how far out of synchrony the clocks can become.

Key Words and Phrases: distributed systems, computer networks, clock synchronization, multiprocess systems

CR Categories: 4.32, 5.29

Introduction

The concept of time is fundamental to our way of thinking. It is derived from the more basic concept of the order in which events occur. We say that something happened at 3:15 if it occurred *after* our clock read 3:15 and *before* it read 3:16. The concept of the temporal ordering of events pervades our thinking about systems. For example, in an airline reservation system we specify that a request for a reservation should be granted if it is made *before* the flight is filled. However, we will see that this concept must be carefully reexamined when considering events in a distributed system.

General permission to make fair use in teaching or research of all or part of this material is granted to individual readers and to nonprofit libraries acting for them provided that ACM's copyright notice is given and that reference is made to the publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Association for Computing Machinery. To otherwise reprint a figure, table, other substantial excerpt, or the entire work requires specific permission as does republication, or systematic or multiple reproduction.

This work was supported by the Advanced Research Projects Agency of the Department of Defense and Rome Air Development Center. It was monitored by Rome Air Development Center under contract number F 30602-76-C-0094.

Author's address: Computer Science Laboratory, SRI International, 333 Ravenswood Ave., Menlo Park CA 94025.

© 1978 ACM 0001-0782/78/0700-0558 $00.75

A distributed system consists of a collection of distinct processes which are spatially separated, and which communicate with one another by exchanging messages. A network of interconnected computers, such as the ARPA net, is a distributed system. A single computer can also be viewed as a distributed system in which the central control unit, the memory units, and the input-output channels are separate processes. A system is distributed if the message transmission delay is not negligible compared to the time between events in a single process.

We will concern ourselves primarily with systems of spatially separated computers. However, many of our remarks will apply more generally. In particular, a multiprocessing system on a single computer involves problems similar to those of a distributed system because of the unpredictable order in which certain events can occur.

In a distributed system, it is sometimes impossible to say that one of two events occurred first. The relation "happened before" is therefore only a partial ordering of the events in the system. We have found that problems often arise because people are not fully aware of this fact and its implications.

In this paper, we discuss the partial ordering defined by the "happened before" relation, and give a distributed algorithm for extending it to a consistent total ordering of all the events. This algorithm can provide a useful mechanism for implementing a distributed system. We illustrate its use with a simple method for solving synchronization problems. Unexpected, anomalous behavior can occur if the ordering obtained by this algorithm differs from that perceived by the user. This can be avoided by introducing real, physical clocks. We describe a simple method for synchronizing these clocks, and derive an upper bound on how far out of synchrony they can drift.

The Partial Ordering

Most people would probably say that an event a happened before an event b if a happened at an earlier time than b. They might justify this definition in terms of physical theories of time. However, if a system is to meet a specification correctly, then that specification must be given in terms of events observable within the system. If the specification is in terms of physical time, then the system must contain real clocks. Even if it does contain real clocks, there is still the problem that such clocks are not perfectly accurate and do not keep precise physical time. We will therefore define the "happened before" relation without using physical clocks.

We begin by defining our system more precisely. We assume that the system is composed of a collection of processes. Each process consists of a sequence of events. Depending upon the application, the execution of a subprogram on a computer could be one event, or the execution of a single machine instruction could be one

Fig. 1.

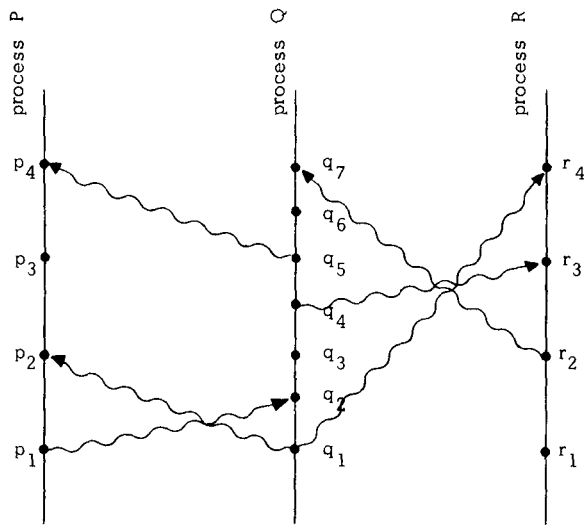
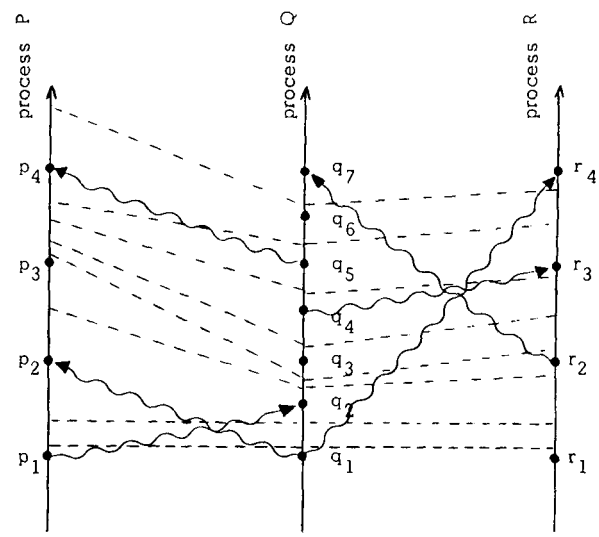


Fig. 2.



event. We are assuming that the events of a process form a sequence, where a occurs before b in this sequence if a happens before b. In other words, a single process is defined to be a set of events with an a priori total ordering. This seems to be what is generally meant by a process.[1] It would be trivial to extend our definition to allow a process to split into distinct subprocesses, but we will not bother to do so.

We assume that sending or receiving a message is an event in a process. We can then define the "happened before" relation, denoted by "$\rightarrow$", as follows.

Definition. The relation "$\rightarrow$" on the set of events of a system is the smallest relation satisfying the following three conditions: (1) If a and b are events in the same process, and a comes before b, then $a \rightarrow b$. (2) If a is the sending of a message by one process and b is the receipt of the same message by another process, then $a \rightarrow b$. (3) If $a \rightarrow b$ and $b \rightarrow c$ then $a \rightarrow c$. Two distinct events a and b are said to be *concurrent* if $a \not\rightarrow b$ and $b \not\rightarrow a$.

We assume that $a \not\rightarrow a$ for any event a. (Systems in which an event can happen before itself do not seem to be physically meaningful.) This implies that $\rightarrow$ is an irreflexive partial ordering on the set of all events in the system.

It is helpful to view this definition in terms of a "space-time diagram" such as Figure 1. The horizontal direction represents space, and the vertical direction represents time—later times being higher than earlier ones. The dots denote events, the vertical lines denote processes, and the wavy lines denote messages.[2] It is easy to see that $a \rightarrow b$ means that one can go from a to b in the diagram by moving forward in time along process and message lines. For example, we have $p_1 \rightarrow r_4$ in Figure 1.

Another way of viewing the definition is to say that $a \rightarrow b$ means that it is possible for event a to causally affect event b. Two events are concurrent if neither can causally affect the other. For example, events p_3 and q_3 of Figure 1 are concurrent. Even though we have drawn the diagram to imply that q_3 occurs at an earlier physical time than p_3, process P cannot know what process Q did at q_3 until it receives the message at p_4. (Before event p_4, P could at most know what Q was *planning* to do at q_3.)

This definition will appear quite natural to the reader familiar with the invariant space-time formulation of special relativity, as described for example in [1] or the first chapter of [2]. In relativity, the ordering of events is defined in terms of messages that *could* be sent. However, we have taken the more pragmatic approach of only considering messages that actually *are* sent. We should be able to determine if a system performed correctly by knowing only those events which *did* occur, without knowing which events *could* have occurred.

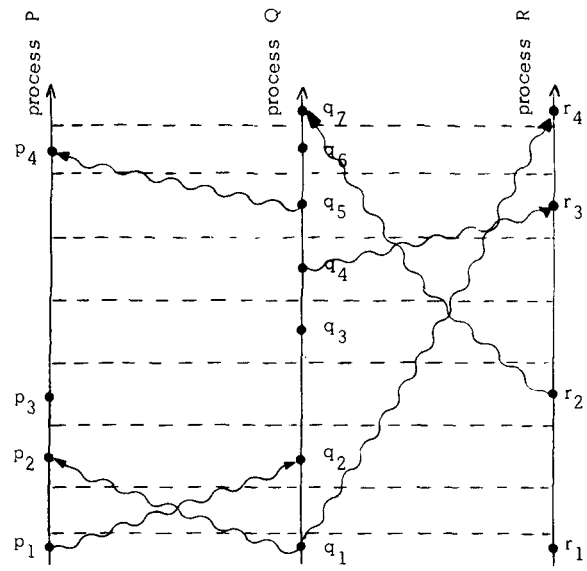
Logical Clocks

We now introduce clocks into the system. We begin with an abstract point of view in which a clock is just a way of assigning a number to an event, where the number is thought of as the time at which the event occurred. More precisely, we define a clock C_i for each process P_i to be a function which assigns a number $C_i\langle a\rangle$ to any event a in that process. The entire system of clocks is represented by the function C which assigns to any event b the number $C\langle b\rangle$, where $C\langle b\rangle = C_j\langle b\rangle$ if b is an event in process P_j. For now, we make no assumption about the relation of the numbers $C_i\langle a\rangle$ to physical time, so we can think of the clocks C_i as logical rather than physical clocks. They may be implemented by counters with no actual timing mechanism.

[1] The choice of what constitutes an event affects the ordering of events in a process. For example, the receipt of a message might denote the setting of an interrupt bit in a computer, or the execution of a subprogram to handle that interrupt. Since interrupts need not be handled in the order that they occur, this choice will affect the ordering of a process' message-receiving events.

[2] Observe that messages may be received out of order. We allow the sending of several messages to be a single event, but for convenience we will assume that the receipt of a single message does not coincide with the sending or receipt of any other message.

Fig. 3.



We now consider what it means for such a system of clocks to be correct. We cannot base our definition of correctness on physical time, since that would require introducing clocks which keep physical time. Our definition must be based on the order in which events occur. The strongest reasonable condition is that if an event a occurs before another event b, then a should happen at an earlier time than b. We state this condition more formally as follows.

Clock Condition. For any events a, b:

$$\text{if } a \rightarrow b \text{ then } C\langle a\rangle < C\langle b\rangle.$$

Note that we cannot expect the converse condition to hold as well, since that would imply that any two concurrent events must occur at the same time. In Figure 1, p_2 and p_3 are both concurrent with q_3, so this would mean that they both must occur at the same time as q_3, which would contradict the Clock Condition because $p_2 \rightarrow p_3$.

It is easy to see from our definition of the relation "$\rightarrow$" that the Clock Condition is satisfied if the following two conditions hold.

C1. If a and b are events in process P_i, and a comes before b, then $C_i\langle a\rangle < C_i\langle b\rangle$.

C2. If a is the sending of a message by process P_i and b is the receipt of that message by process P_j, then $C_i\langle a\rangle < C_j\langle b\rangle$.

Let us consider the clocks in terms of a space-time diagram. We imagine that a process' clock "ticks" through every number, with the ticks occurring between the process' events. For example, if a and b are consecutive events in process P_i with $C_i\langle a\rangle = 4$ and $C_i\langle b\rangle = 7$, then clock ticks 5, 6, and 7 occur between the two events. We draw a dashed "tick line" through all the like-numbered ticks of the different processes. The space-time diagram of Figure 1 might then yield the picture in Figure 2. Condition C1 means that there must be a tick line between any two events on a process line, and condition C2 means that every message line must cross a tick line. From the pictorial meaning of $\rightarrow$, it is easy to see why these two conditions imply the Clock Condition.

We can consider the tick lines to be the time coordinate lines of some Cartesian coordinate system on space-time. We can redraw Figure 2 to straighten these coordinate lines, thus obtaining Figure 3. Figure 3 is a valid alternate way of representing the same system of events as Figure 2. Without introducing the concept of physical time into the system (which requires introducing physical clocks), there is no way to decide which of these pictures is a better representation.

The reader may find it helpful to visualize a two-dimensional spatial network of processes, which yields a three-dimensional space-time diagram. Processes and messages are still represented by lines, but tick lines become two-dimensional surfaces.

Let us now assume that the processes are algorithms, and the events represent certain actions during their execution. We will show how to introduce clocks into the processes which satisfy the Clock Condition. Process P_i's clock is represented by a register C_i, so that $C_i\langle a\rangle$ is the value contained by C_i during the event a. The value of C_i will change between events, so changing C_i does not itself constitute an event.

To guarantee that the system of clocks satisfies the Clock Condition, we will insure that it satisfies conditions C1 and C2. Condition C1 is simple; the processes need only obey the following implementation rule:

IR1. Each process P_i increments C_i between any two successive events.

To meet condition C2, we require that each message m contain a *timestamp* T_m which equals the time at which the message was sent. Upon receiving a message time-stamped T_m, a process must advance its clock to be later than T_m. More precisely, we have the following rule.

IR2. (a) If event a is the sending of a message m by process P_i, then the message m contains a timestamp $T_m = C_i\langle a\rangle$. (b) Upon receiving a message m, process P_j sets C_j greater than or equal to its present value and greater than T_m.

In IR2(b) we consider the event which represents the receipt of the message m to occur after the setting of C_j. (This is just a notational nuisance, and is irrelevant in any actual implementation.) Obviously, IR2 insures that C2 is satisfied. Hence, the simple implementation rules IR1 and IR2 imply that the Clock Condition is satisfied, so they guarantee a correct system of logical clocks.

Ordering the Events Totally

We can use a system of clocks satisfying the Clock Condition to place a total ordering on the set of all system events. We simply order the events by the times

at which they occur. To break ties, we use any arbitrary total ordering $<$ of the processes. More precisely, we define a relation $\Rightarrow$ as follows: if a is an event in process P_i and b is an event in process P_j, then $a \Rightarrow b$ if and only if either (i) $C_i\langle a\rangle < C_j\langle b\rangle$ or (ii) $C_i\langle a\rangle = C_j\langle b\rangle$ and $P_i < P_j$. It is easy to see that this defines a total ordering, and that the Clock Condition implies that if $a \rightarrow b$ then $a \Rightarrow b$. In other words, the relation $\Rightarrow$ is a way of completing the "happened before" partial ordering to a total ordering.[3]

The ordering $\Rightarrow$ depends upon the system of clocks C_i, and is not unique. Different choices of clocks which satisfy the Clock Condition yield different relations $\Rightarrow$. Given any total ordering relation $\Rightarrow$ which extends $\rightarrow$, there is a system of clocks satisfying the Clock Condition which yields that relation. It is only the partial ordering $\rightarrow$ which is uniquely determined by the system of events.

Being able to totally order the events can be very useful in implementing a distributed system. In fact, the reason for implementing a correct system of logical clocks is to obtain such a total ordering. We will illustrate the use of this total ordering of events by solving the following version of the mutual exclusion problem. Consider a system composed of a fixed collection of processes which share a single resource. Only one process can use the resource at a time, so the processes must synchronize themselves to avoid conflict. We wish to find an algorithm for granting the resource to a process which satisfies the following three conditions: (I) A process which has been granted the resource must release it before it can be granted to another process. (II) Different requests for the resource must be granted in the order in which they are made. (III) If every process which is granted the resource eventually releases it, then every request is eventually granted.

We assume that the resource is initially granted to exactly one process.

These are perfectly natural requirements. They precisely specify what it means for a solution to be correct.[4] Observe how the conditions involve the ordering of events. Condition II says nothing about which of two concurrently issued requests should be granted first.

It is important to realize that this is a nontrivial problem. Using a central scheduling process which grants requests in the order they are received will not work, unless additional assumptions are made. To see this, let P_0 be the scheduling process. Suppose P_1 sends a request to P_0 and then sends a message to P_2. Upon receiving the latter message, P_2 sends a request to P_0. It is possible for P_2's request to reach P_0 before P_1's request does. Condition II is then violated if P_2's request is granted first.

To solve the problem, we implement a system of clocks with rules IR1 and IR2, and use them to define a total ordering $\Rightarrow$ of all events. This provides a total ordering of all request and release operations. With this ordering, finding a solution becomes a straightforward exercise. It just involves making sure that each process learns about all other processes' operations.

To simplify the problem, we make some assumptions. They are not essential, but they are introduced to avoid distracting implementation details. We assume first of all that for any two processes P_i and P_j, the messages sent from P_i to P_j are received in the same order as they are sent. Moreover, we assume that every message is eventually received. (These assumptions can be avoided by introducing message numbers and message acknowledgment protocols.) We also assume that a process can send messages directly to every other process.

Each process maintains its own *request queue* which is never seen by any other process. We assume that the request queues initially contain the single message T_0:P_0 *requests resource*, where P_0 is the process initially granted the resource and T_0 is less than the initial value of any clock.

The algorithm is then defined by the following five rules. For convenience, the actions defined by each rule are assumed to form a single event.

1. To request the resource, process P_i sends the message T_m:P_i *requests resource* to every other process, and puts that message on its request queue, where T_m is the timestamp of the message.

2. When process P_j receives the message T_m:P_i *requests resource*, it places it on its request queue and sends a (timestamped) acknowledgment message to P_i.[5]

3. To release the resource, process P_i removes any T_m:P_i *requests resource* message from its request queue and sends a (timestamped) P_i *releases resource* message to every other process.

4. When process P_j receives a P_i *releases resource* message, it removes any T_m:P_i *requests resource* message from its request queue.

5. Process P_i is granted the resource when the following two conditions are satisfied: (i) There is a T_m:P_i *requests resource* message in its request queue which is ordered before any other request in its queue by the relation $\Rightarrow$. (To define the relation "$\Rightarrow$" for messages, we identify a message with the event of sending it.) (ii) P_i has received a message from every other process timestamped later than T_m.[6]

Note that conditions (i) and (ii) of rule 5 are tested locally by P_i.

It is easy to verify that the algorithm defined by these rules satisfies conditions I–III. First of all, observe that condition (ii) of rule 5, together with the assumption that messages are received in order, guarantees that P_i has learned about all requests which preceded its current

[3] The ordering $<$ establishes a priority among the processes. If a "fairer" method is desired, then $<$ can be made a function of the clock value. For example, if $C_i\langle a\rangle = C_j\langle b\rangle$ and $j < i$, then we can let $a \Rightarrow b$ if $j < C_i\langle a\rangle \bmod N \leq i$, and $b \Rightarrow a$ otherwise; where N is the total number of processes.

[4] The term "eventually" should be made precise, but that would require too long a diversion from our main topic.

[5] This acknowledgment message need not be sent if P_j has already sent a message to P_i timestamped later than T_m.

[6] If $P_i < P_j$, then P_i need only have received a message timestamped $\geq T_m$ from P_j.

request. Since rules 3 and 4 are the only ones which delete messages from the request queue, it is then easy to see that condition I holds. Condition II follows from the fact that the total ordering $\Rightarrow$ extends the partial ordering $\rightarrow$. Rule 2 guarantees that after P_i requests the resource, condition (ii) of rule 5 will eventually hold. Rules 3 and 4 imply that if each process which is granted the resource eventually releases it, then condition (i) of rule 5 will eventually hold, thus proving condition III.

This is a distributed algorithm. Each process independently follows these rules, and there is no central synchronizing process or central storage. This approach can be generalized to implement any desired synchronization for such a distributed multiprocess system. The synchronization is specified in terms of a *State Machine*, consisting of a set $\boldsymbol{C}$ of possible commands, a set $\boldsymbol{S}$ of possible states, and a function $\boldsymbol{e}: \boldsymbol{C} \times \boldsymbol{S} \rightarrow \boldsymbol{S}$. The relation $\boldsymbol{e}(C, S) = S'$ means that executing the command C with the machine in state S causes the machine state to change to S'. In our example, the set $\boldsymbol{C}$ consists of all the commands P_i *requests resource* and P_i *releases resource*, and the state consists of a queue of waiting *request* commands, where the request at the head of the queue is the currently granted one. Executing a *request* command adds the request to the tail of the queue, and executing a *release* command removes a command from the queue.[7]

Each process independently simulates the execution of the State Machine, using the commands issued by all the processes. Synchronization is achieved because all processes order the commands according to their timestamps (using the relation $\Rightarrow$), so each process uses the same sequence of commands. A process can execute a command timestamped T when it has learned of all commands issued by all other processes with timestamps less than or equal to T. The precise algorithm is straightforward, and we will not bother to describe it.

This method allows one to implement any desired form of multiprocess synchronization in a distributed system. However, the resulting algorithm requires the active participation of all the processes. A process must know all the commands issued by other processes, so that the failure of a single process will make it impossible for any other process to execute State Machine commands, thereby halting the system.

The problem of failure is a difficult one, and it is beyond the scope of this paper to discuss it in any detail. We will just observe that the entire concept of failure is only meaningful in the context of physical time. Without physical time, there is no way to distinguish a failed process from one which is just pausing between events. A user can tell that a system has "crashed" only because he has been waiting too long for a response. A method which works despite the failure of individual processes or communication lines is described in [3].

[7] If each process does not strictly alternate *request* and *release* commands, then executing a *release* command could delete zero, one, or more than one request from the queue.

Anomalous Behavior

Our resource scheduling algorithm ordered the requests according to the total ordering $\Rightarrow$. This permits the following type of "anomalous behavior." Consider a nationwide system of interconnected computers. Suppose a person issues a request A on a computer A, and then telephones a friend in another city to have him issue a request B on a different computer B. It is quite possible for request B to receive a lower timestamp and be ordered before request A. This can happen because the system has no way of knowing that A actually preceded B, since that precedence information is based on messages external to the system.

Let us examine the source of the problem more closely. Let $\mathcal{S}$ be the set of all system events. Let us introduce a set of events which contains the events in $\mathcal{S}$ together with all other relevant external events, such as the phone calls in our example. Let $\rightarrow$ denote the "happened before" relation for $\underline{\mathcal{S}}$. In our example, we had A $\rightarrow$ B, but A $\not\rightarrow$ B. It is obvious that no algorithm based entirely upon events in $\mathcal{S}$, and which does not relate those events in any way with the other events in $\underline{\mathcal{S}}$, can guarantee that request A is ordered before request B.

There are two possible ways to avoid such anomalous behavior. The first way is to explicitly introduce into the system the necessary information about the ordering $\rightarrow$. In our example, the person issuing request A could receive the timestamp T_A of that request from the system. When issuing request B, his friend could specify that B be given a timestamp later than T_A. This gives the user the responsibility for avoiding anomalous behavior.

The second approach is to construct a system of clocks which satisfies the following condition.

Strong Clock Condition. For any events a, b in $\mathcal{S}$:

$$\text{if } a \rightarrow b \text{ then } C\langle a\rangle < C\langle b\rangle.$$

This is stronger than the ordinary Clock Condition because $\rightarrow$ is a stronger relation than $\rightarrow$. It is not in general satisfied by our logical clocks.

Let us identify $\underline{\mathcal{S}}$ with some set of "real" events in physical space-time, and let $\rightarrow$ be the partial ordering of events defined by special relativity. One of the mysteries of the universe is that it is possible to construct a system of physical clocks which, running quite independently of one another, will satisfy the Strong Clock Condition. We can therefore use physical clocks to eliminate anomalous behavior. We now turn our attention to such clocks.

Physical Clocks

Let us introduce a physical time coordinate into our space-time picture, and let $C_i(t)$ denote the reading of the clock C_i at physical time t.[8] For mathematical con-

[8] We will assume a Newtonian space-time. If the relative motion of the clocks or gravitational effects are not negligible, then $C_i(t)$ must be deduced from the actual clock reading by transforming from proper time to the arbitrarily chosen time coordinate.

venience, we assume that the clocks run continuously rather than in discrete "ticks." (A discrete clock can be thought of as a continuous one in which there is an error of up to $\frac{1}{2}$ "tick" in reading it.) More precisely, we assume that $C_i(t)$ is a continuous, differentiable function of t except for isolated jump discontinuities where the clock is reset. Then $dC_i(t)/dt$ represents the rate at which the clock is running at time t.

In order for the clock C_i to be a true physical clock, it must run at approximately the correct rate. That is, we must have $dC_i(t)/dt \approx 1$ for all t. More precisely, we will assume that the following condition is satisfied:

PC1. There exists a constant $\kappa \ll 1$ such that for all i: $|dC_i(t)/dt - 1| < \kappa$.

For typical crystal controlled clocks, $\kappa \leq 10^{-6}$.

It is not enough for the clocks individually to run at approximately the correct rate. They must be synchronized so that $C_i(t) \approx C_j(t)$ for all i, j, and t. More precisely, there must be a sufficiently small constant ϵ so that the following condition holds:

PC2. For all i, j: $|C_i(t) - C_j(t)| < \epsilon$.

If we consider vertical distance in Figure 2 to represent physical time, then PC2 states that the variation in height of a single tick line is less than ϵ.

Since two different clocks will never run at exactly the same rate, they will tend to drift further and further apart. We must therefore devise an algorithm to insure that PC2 always holds. First, however, let us examine how small κ and ϵ must be to prevent anomalous behavior. We must insure that the system $\underline{\mathcal{S}}$ of relevant physical events satisfies the Strong Clock Condition. We assume that our clocks satisfy the ordinary Clock Condition, so we need only require that the Strong Clock Condition holds when a and b are events in $\underline{\mathcal{S}}$ with $a \not\rightarrow b$. Hence, we need only consider events occurring in different processes.

Let μ be a number such that if event a occurs at physical time t and event b in another process satisfies $a \rightarrow b$, then b occurs later than physical time $t + \mu$. In other words, μ is less than the shortest transmission time for interprocess messages. We can always choose μ equal to the shortest distance between processes divided by the speed of light. However, depending upon how messages in $\underline{\mathcal{S}}$ are transmitted, μ could be significantly larger.

To avoid anomalous behavior, we must make sure that for any i, j, and t: $C_i(t + \mu) - C_j(t) > 0$. Combining this with PC1 and 2 allows us to relate the required smallness of κ and ϵ to the value of μ as follows. We assume that when a clock is reset, it is always set forward and never back. (Setting it back could cause C1 to be violated.) PC1 then implies that $C_i(t + \mu) - C_i(t) > (1 - \kappa)\mu$. Using PC2, it is then easy to deduce that $C_i(t + \mu) - C_j(t) > 0$ if the following inequality holds:

$$\epsilon/(1 - \kappa) \leq \mu.$$

This inequality together with PC1 and PC2 implies that anomalous behavior is impossible.

We now describe our algorithm for insuring that PC2 holds. Let m be a message which is sent at physical time t and received at time t'. We define $\nu_m = t' - t$ to be the *total delay* of the message m. This delay will, of course, not be known to the process which receives m. However, we assume that the receiving process knows some *minimum delay* $\mu_m \geq 0$ such that $\mu_m \leq \nu_m$. We call $\xi_m = \nu_m - \mu_m$ the *unpredictable delay* of the message.

We now specialize rules IR1 and 2 for our physical clocks as follows:

IR1′. For each i, if P_i does not receive a message at physical time t, then C_i is differentiable at t and $dC_i(t)/dt > 0$.

IR2′. (a) If P_i sends a message m at physical time t, then m contains a timestamp $T_m = C_i(t)$. (b) Upon receiving a message m at time t', process P_j sets $C_j(t')$ equal to maximum $(C_j(t' - 0), T_m + \mu_m)$.[9]

Although the rules are formally specified in terms of the physical time parameter, a process only needs to know its own clock reading and the timestamps of messages it receives. For mathematical convenience, we are assuming that each event occurs at a precise instant of physical time, and different events in the same process occur at different times. These rules are then specializations of rules IR1 and IR2, so our system of clocks satisfies the Clock Condition. The fact that real events have a finite duration causes no difficulty in implementing the algorithm. The only real concern in the implementation is making sure that the discrete clock ticks are frequent enough so C1 is maintained.

We now show that this clock synchronizing algorithm can be used to satisfy condition PC2. We assume that the system of processes is described by a directed graph in which an arc from process P_i to process P_j represents a communication line over which messages are sent directly from P_i to P_j. We say that a message is sent over this arc every τ seconds if for any t, P_i sends at least one message to P_j between physical times t and $t + \tau$. The *diameter* of the directed graph is the smallest number d such that for any pair of distinct processes P_j, P_k, there is a path from P_j to P_k having at most d arcs.

In addition to establishing PC2, the following theorem bounds the length of time it can take the clocks to become synchronized when the system is first started.

THEOREM. Assume a strongly connected graph of processes with diameter d which always obeys rules IR1′ and IR2′. Assume that for any message m, $\mu_m \leq \mu$ for some constant μ, and that for all $t \geq t_0$: (a) PC1 holds. (b) There are constants τ and ξ such that every τ seconds a message with an unpredictable delay less than ξ is sent over every arc. Then PC2 is satisfied with $\epsilon \approx d(2\kappa\tau + \xi)$ for all $t \gtrsim t_0 + \tau d$, where the approximations assume $\mu + \xi \ll \tau$.

The proof of this theorem is surprisingly difficult, and is given in the Appendix. There has been a great deal of work done on the problem of synchronizing physical clocks. We refer the reader to [4] for an intro-

[9] $C_j(t' - 0) = \lim_{\delta \to 0} C_j(t' - |\delta|)$.

duction to the subject. The methods described in the literature are useful for estimating the message delays μ_m and for adjusting the clock frequencies dC_i/dt (for clocks which permit such an adjustment). However, the requirement that clocks are never set backwards seems to distinguish our situation from ones previously studied, and we believe this theorem to be a new result.

Conclusion

We have seen that the concept of "happening before" defines an invariant partial ordering of the events in a distributed multiprocess system. We described an algorithm for extending that partial ordering to a somewhat arbitrary total ordering, and showed how this total ordering can be used to solve a simple synchronization problem. A future paper will show how this approach can be extended to solve any synchronization problem.

The total ordering defined by the algorithm is somewhat arbitrary. It can produce anomalous behavior if it disagrees with the ordering perceived by the system's users. This can be prevented by the use of properly synchronized physical clocks. Our theorem showed how closely the clocks can be synchronized.

In a distributed system, it is important to realize that the order in which events occur is only a partial ordering. We believe that this idea is useful in understanding any multiprocess system. It should help one to understand the basic problems of multiprocessing independently of the mechanisms used to solve them.

Appendix

Proof of the Theorem

For any i and t, let us define C_i^t to be a clock which is set equal to C_i at time t and runs at the same rate as C_i, but is never reset. In other words,

$$C_i^t(t') = C_i(t) + \int_t^{t'} [dC_i(t)/dt]dt \tag{1}$$

for all $t' \geq t$. Note that

$$C_i(t') \geq C_i^t(t') \text{ for all } t' \geq t. \tag{2}$$

Suppose process P_1 at time t_1 sends a message to process P_2 which is received at time t_2 with an unpredictable delay $\leq \xi$, where $t_0 \leq t_1 \leq t_2$. Then for all $t \geq t_2$ we have:

$$\begin{aligned} C_2^{t_2}(t) &\geq C_2^{t_2}(t_2) + (1-\kappa)(t - t_2) && \text{[by (1) and PC1]} \\ &\geq C_1(t_1) + \mu_m + (1-\kappa)(t - t_2) && \text{[by IR2' (b)]} \\ &= C_1(t_1) + (1-\kappa)(t - t_1) - [(t_2 - t_1) - \mu_m] + \kappa(t_2 - t_1) \\ &\geq C_1(t_1) + (1-\kappa)(t - t_1) - \xi. \end{aligned}$$

Hence, with these assumptions, for all $t \geq t_2$ we have:

$$C_2^{t_2}(t) \geq C_1(t_1) + (1-\kappa)(t - t_1) - \xi. \tag{3}$$

Now suppose that for $i = 1, \ldots, n$ we have $t_i \leq t_i' < t_{i+1}$, $t_0 \leq t_1$, and that at time t_i' process P_i sends a message to process P_{i+1} which is received at time t_{i+1} with an unpredictable delay less than ξ. Then repeated application of the inequality (3) yields the following result for $t \geq t_{n+1}$.

$$C_{n+1}^{t_{n+1}}(t) \geq C_1(t_1') + (1-\kappa)(t - t_1') - n\xi. \tag{4}$$

From PC1, IR1' and 2' we deduce that

$$C_1(t_1') \geq C_1(t_1) + (1-\kappa)(t_1' - t_1).$$

Combining this with (4) and using (2), we get

$$C_{n+1}(t) \geq C_1(t_1) + (1-\kappa)(t - t_1) - n\xi \tag{5}$$

for $t \geq t_{n+1}$.

For any two processes P and P', we can find a sequence of processes $P = P_0, P_1, \ldots, P_{n+1} = P'$, $n \leq d$, with communication arcs from each P_i to P_{i+1}. By hypothesis (b) we can find times t_i, t_i' with $t_i' - t_i \leq \tau$ and $t_{i+1} - t_i' \leq \nu$, where $\nu = \mu + \xi$. Hence, an inequality of the form (5) holds with $n \leq d$ whenever $t \geq t_1 + d(\tau + \nu)$. For any i, j and any t, t_1 with $t_1 \geq t_0$ and $t \geq t_1 + d(\tau + \nu)$ we therefore have:

$$C_i(t) \geq C_j(t_1) + (1-\kappa)(t - t_1) - d\xi. \tag{6}$$

Now let m be any message timestamped T_m, and suppose it is sent at time t and received at time t'. We pretend that m has a clock C_m which runs at a constant rate such that $C_m(t) = t_m$ and $C_m(t') = t_m + \mu_m$. Then $\mu_m \leq t' - t$ implies that $dC_m/dt \leq 1$. Rule IR2' (b) simply sets $C_j(t')$ to *maximum* $(C_j(t' - 0), C_m(t'))$. Hence, clocks are reset only by setting them equal to other clocks.

For any time $t_x \geq t_0 + \mu/(1-\kappa)$, let C_x be the clock having the largest value at time t_x. Since all clocks run at a rate less than $1 + \kappa$, we have for all i and all $t \geq t_x$:

$$C_i(t) \leq C_x(t_x) + (1+\kappa)(t - t_x). \tag{7}$$

We now consider the following two cases: (i) C_x is the clock C_q of process P_q. (ii) C_x is the clock C_m of a message sent at time t_1 by process P_q. In case (i), (7) simply becomes

$$C_i(t) \leq C_q(t_x) + (1+\kappa)(t - t_x). \tag{8i}$$

In case (ii), since $C_m(t_1) = C_q(t_1)$ and $dC_m/dt \leq 1$, we have

$$C_x(t_x) \leq C_q(t_1) + (t_x - t_1).$$

Hence, (7) yields

$$C_i(t) \leq C_q(t_1) + (1+\kappa)(t - t_1). \tag{8ii}$$

Since $t_x \geq t_0 + \mu/(1-\kappa)$, we get

$$\begin{aligned} C_q(t_x - \mu/(1-\kappa)) &\leq C_q(t_x) - \mu && \text{[by PC1]} \\ &\leq C_m(t_x) - \mu && \text{[by choice of } m\text{]} \\ &\leq C_m(t_x) - (t_x - t_1)\mu_m/\nu_m && [\mu_m \leq \mu,\ t_x - t_1 \leq \nu_m] \\ &= T_m && \text{[by definition of } C_m\text{]} \\ &= C_q(t_1) && \text{[by IR2'(a)].} \end{aligned}$$

Hence, $C_q(t_x - \mu/(1-\kappa)) \leq C_q(t_1)$, so $t_x - t_1 \leq \mu/(1-\kappa)$ and thus $t_1 \geq t_0$.

Letting $t_1 = t_x$ in case (i), we can combine (8i) and (8ii) to deduce that for any t, t_x with $t \geq t_x \geq t_0 + \mu/(1 - \kappa)$ there is a process P_q and a time t_1 with $t_x - \mu/(1 - \kappa) \leq t_1 \leq t_x$ such that for all i:

$$C_i(t) \leq C_q(t_1) + (1 + \kappa)(t - t_1). \tag{9}$$

Choosing t and t_x with $t \geq t_x + d(\tau + \nu)$, we can combine (6) and (9) to conclude that there exists a t_1 and a process P_q such that for all i:

$$\begin{aligned} C_q(t_1) + (1 - \kappa)(t - t_1) - d\xi &\leq C_i(t) \\ &\leq C_q(t_1) + (1 + \kappa)(t - t_1) \end{aligned} \tag{10}$$

Letting $t = t_x + d(\tau + \nu)$, we get

$$d(\tau + \nu) \leq t - t_1 \leq d(\tau + \nu) + \mu/(1 - \kappa).$$

Combining this with (10), we get

$$\begin{aligned} C_q(t_1) + (t - t_1) - \kappa d(\tau + \nu) - d\xi \leq C_i(t) \leq C_q(t_1) \\ + (t - t_1) + \kappa[d(\tau + \nu) + \mu/(1 - \kappa)] \end{aligned} \tag{11}$$

Using the hypotheses that $\kappa \ll 1$ and $\mu \leq \nu \ll \tau$, we can rewrite (11) as the following approximate inequality.

$$\begin{aligned} C_q(t_1) + (t - t_1) - d(\kappa\tau + \xi) &\lesssim C_i(t) \\ &\lesssim C_q(t_1) + (t - t_1) + d\kappa\tau. \end{aligned} \tag{12}$$

Since this holds for all i, we get

$$|C_i(t) - C_j(t)| \lesssim d(2\kappa\tau + \xi),$$

and this holds for all $t \gtrsim t_0 + d\tau$. □

Note that relation (11) of the proof yields an exact upper bound for $|C_i(t) - C_j(t)|$ in case the assumption $\mu + \xi \ll \tau$ is invalid. An examination of the proof suggests a simple method for rapidly initializing the clocks, or resynchronizing them if they should go out of synchrony for any reason. Each process sends a message which is relayed to every other process. The procedure can be initiated by any process, and requires less than $2d(\mu + \xi)$ seconds to effect the synchronization, assuming each of the messages has an unpredictable delay less than ξ.

Acknowledgment. The use of timestamps to order operations, and the concept of anomalous behavior are due to Paul Johnson and Robert Thomas.

Received March 1976; revised October 1977

References

1. Schwartz, J.T. *Relativity in Illustrations.* New York U. Press, New York, 1962.
2. Taylor, E.F., and Wheeler, J.A. *Space-Time Physics*, W.H. Freeman, San Francisco, 1966.
3. Lamport, L. The implementation of reliable distributed multiprocess systems. To appear in *Computer Networks.*
4. Ellingson, C., and Kulpinski, R.J. Dissemination of system-time. *IEEE Trans. Comm. Com-23*, 5 (May 1973), 605–624.

Programming Languages

J. J. Horning
Editor

Shallow Binding in Lisp 1.5

Henry G. Baker, Jr.
Massachusetts Institute of Technology

Shallow binding is a scheme which allows the value of a variable to be accessed in a bounded amount of computation. An elegant model for shallow binding in Lisp 1.5 is presented in which context-switching is an environment tree transformation called rerooting. Rerooting is completely general and reversible, and is optional in the sense that a Lisp 1.5 interpreter will operate correctly whether or not rerooting is invoked on every context change. Since rerooting leaves assoc [*v*, *a*] invariant, for all variables *v* and all environments *a*, the programmer can have access to a rerooting primitive, shallow[], which gives him dynamic control over whether accesses are shallow or deep, and which affects only the speed of execution of a program, not its semantics. In addition, multiple processes can be active in the same environment structure, so long as rerooting is an indivisible operation. Finally, the concept of rerooting is shown to combine the concept of shallow binding in Lisp with Dijkstra's display for Algol and hence is a general model for shallow binding.

Key Words and Phrases: Lisp 1.5, environment trees, FUNARG's, shallow binding, deep binding, multiprogramming, Algol display

CR Categories: 4.13, 4.22, 4.32

General permission to make fair use in teaching or research of all or part of this material is granted to individual readers and to nonprofit libraries acting for them provided that ACM's copyright notice is given and that reference is made to the publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Association for Computing Machinery. To otherwise reprint a figure, table, other substantial excerpt, or the entire work requires specific permission as does republication, or systematic or multiple reproduction.

This research was supported by the Advanced Research Projects Agency of the Department of Defense and was monitored by the Office of Naval Research under contract number N00014-75-C-0522.

Author's present address: Computer Science Department, University of Rochester, Rochester, NY 14627.

© 1978 ACM 0001-0782/78/0700-0565 $00.75

Chapter 2

Distributed Snapshots: Determining Global States of Distributed Systems

Authors:
K. Mani Chandy, Leslie Lamport

Publication year: 1985

Distributed Snapshots: Determining Global States of Distributed Systems

K. MANI CHANDY
University of Texas at Austin
and
LESLIE LAMPORT
Stanford Research Institute

This paper presents an algorithm by which a process in a distributed system determines a global state of the system during a computation. Many problems in distributed systems can be cast in terms of the problem of detecting global states. For instance, the global state detection algorithm helps to solve an important class of problems: stable property detection. A stable property is one that persists: once a stable property becomes true it remains true thereafter. Examples of stable properties are "computation has terminated," "the system is deadlocked" and "all tokens in a token ring have disappeared." The stable property detection problem is that of devising algorithms to detect a given stable property. Global state detection can also be used for checkpointing.

Categories and Subject Descriptors: C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*distributed applications*; *distributed databases*; *network operating systems*; D.4.1 [**Operating Systems**]: Process Management—*concurrency*; *deadlocks*; *multiprocessing/multiprogramming*; *mutual exclusion*; *scheduling*; *synchronization*; D.4.5 [**Operating Systems**]: Reliability—*backup procedures*; *checkpoint/restart*; *fault-tolerance*; *verification*

General Terms: Algorithms

Additional Key Words and Phrases: Global States, Distributed deadlock detection, distributed systems, message communication systems

1. INTRODUCTION

This paper presents algorithms by which a process in a distributed system can determine a global state of the system during a computation. Processes in a distributed system communicate by sending and receiving messages. A process can record its own state and the messages it sends and receives; *it can record nothing else.* To determine a global system state, a process p must enlist the

This work was supported in part by the Air Force Office of Scientific Research under Grant AFOSR 81-0205 and in part by the National Science Foundation under Grant MCS 81-04459.
Authors' addresses: K. M. Chandy, Department of Computer Sciences, University of Texas at Austin, Austin, TX 78712; L. Lamport, Stanford Research Institute, Menlo Park, CA 94025.
Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
© 1985 ACM 0734-2071/85/0200-0063 $00.75

cooperation of other processes that must record their own local states and send the recorded local states to p. All processes cannot record their local states at precisely the same instant unless they have access to a common clock. We assume that processes do not share clocks or memory. The problem is to devise algorithms by which processes record their own states and the states of communication channels so that the set of process and channel states recorded form a global system state. The global-state-detection algorithm is to be superimposed on the underlying computation: it must run concurrently with, but not alter, this underlying computation.

The state-detection algorithm plays the role of a group of photographers observing a panoramic, dynamic scene, such as a sky filled with migrating birds—a scene so vast that it cannot be captured by a single photograph. The photographers must take several snapshots and piece the snapshots together to form a picture of the overall scene. The snapshots cannot all be taken at precisely the same instant because of synchronization problems. Furthermore, the photographers should not disturb the process that is being photographed; for instance, they cannot get all the birds in the heavens to remain motionless while the photographs are taken. Yet, the composite picture should be meaningful. The problem before us is to define "meaningful" and then to determine how the photographs should be taken.

We now describe an important class of problems that can be solved with the global-state-detection algorithm. Let y be a predicate function defined on the global states of a distributed system D; that is, $y(S)$ is true or false for a global state S of D. The predicate y is said to be a *stable property* of D if $y(S)$ implies $y(S')$ for all global states S' of D reachable from global state S of D. In other words, if y is a stable property and y is true at a point in a computation of D, then y is true at all later points in that computation. Examples of stable properties are "computation has terminated," "the system is deadlocked," and "all tokens in a token ring have disappeared."

Several distributed-system problems can be formulated as the general problem of devising an algorithm by which a process in a distributed system can determine whether a stable property y of the system holds. Deadlock detection [2, 5, 8, 9, 11] and termination detection [1, 4, 10] are special cases of the stable-property detection problem. Details of the algorithm are presented later. The basic idea of the algorithm is that a global state S of the system is determined and $y(S)$ is computed to see if the stable property y holds.

Several algorithms for solving deadlock and termination problems by determining the global states of distributed systems have been published. Gligor and Shattuck [5] state that many of the published algorithms are incorrect and impractical. A reason for the incorrect or impractical algorithms may be that the relationships among local process states, global system states, and points in a distributed computation are not well understood. One of the contributions of this paper is to define these relationships.

Many distributed algorithms are structured as a sequence of phases, where each phase consists of a transient part in which useful work is done, followed by a stable part in which the system cycles endlessly and uselessly. The presence of stable behavior indicates the end of a phase. A phase is similar to a series of

iterations in a sequential program, which are repeated until successive iterations produce no change, that is, stability is attained. Stability must be detected so that one phase can be terminated and the next phase initiated [10]. The termination of a computational phase is not identical to the termination of a computation. When a computation terminates, all activities cease—messages are not sent and process states do not change. There may be activity during the stable behavior that indicates the end of a computational phase—messages may be sent and received, and processes may change state, but this activity serves no purpose other than to signal the end of a phase. In this paper, we are concerned with the detection of stable system properties; the cessation of activity is only one example of a stable property.

Strictly speaking, properties such as "the system is deadlocked" are not stable if the deadlock is "broken" and computation is reinitiated. However, to keep exposition simple, we shall partition the overall problem into the problems of (1) detecting the termination of one phase (and informing all processes that a phase has ended) and (2) initiating a new phase. The following is a stable property: "the kth computational phase has terminated," $k = 1, 2, \ldots$. Hence, the methods presented in this paper are applicable to detecting the termination of the kth phase for a given k.

In this paper we restrict attention to the problem of detecting stable properties. The problem of initiating the next phase of computation is not considered here because the solution to that problem varies significantly depending on the application, being different for database deadlock detection than for detecting the termination of a diffusing computation.

We have to present our algorithms in terms of a model of a system. The model chosen is not important in itself; we could have couched our discussion in terms of other models. We shall describe our model informally and only to the level of detail necessary to make the algorithms clear.

2. MODEL OF A DISTRIBUTED SYSTEM

A distributed system consists of a finite set of processes and a finite set of channels. It is described by a labeled, directed graph in which the vertices represent processes and the edges represent channels. Figure 1 is an example.

Channels are assumed to have infinite buffers, to be error-free, and to deliver messages in the order sent. (The infinite buffer assumption is made for ease of exposition: bounded buffers may be assumed provided there exists a proof that no process attempts to add a message to a full buffer.) The delay experienced by a message in a channel is arbitrary but finite. The sequence of messages received along a channel is an initial subsequence of the sequence of messages sent along the channel. The *state* of a channel is the sequence of messages sent along the channel, excluding the messages received along the channel.

A process is defined by a set of states, an initial state (from this set), and a set of events. An event e in a process p is an atomic action that may change the state of p itself and the state of *at most one* channel c incident on p: the state of c may be changed by the sending of a message along c (if c is directed away from p) or the receipt of a message along c (if c is directed towards p). An event e is defined by (1) the process p in which the event occurs, (2) the state s of p immediately

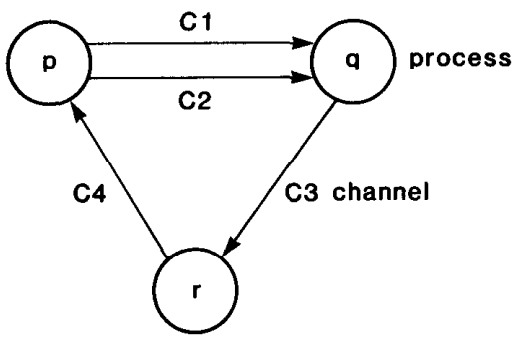


Fig. 1. A distributed system with processes p, q, and r and channels c1, c2, c3, and c4.

before the event, (3) the state s' of p immediately after the event, (4) the channel c (if any) whose state is altered by the event, and (5) the message M, if any, sent along c (if c is a channel directed away from p) or received along c (if c is directed towards p). We define e by the 5-tuple $\langle p, s, s', M, c\rangle$, where M and c are a special symbol, *null*, if the occurrence of e does not change the state of any channel.

A global state of a distributed system is a set of component process and channel states: the *initial* global state is one in which the state of each process is its initial state and the state of each channel is the empty sequence. The occurrence of an event may change the global state. Let $e = \langle p, s, s', M, c\rangle$ we say *e can occur in* global state S if and only if (1) the state of process p in global state S is s and (2) if c is a channel directed towards p, then the state of c in global state S is a sequence of messages with M at its head. We define a function *next*, where $next(S, e)$ is the global state immediately after the occurrence of event e in global state S. The value of $next(S, e)$ is defined only if event e can occur in global state S, in which case $next(S, e)$ is the global state identical to S except that: (1) the state of p in $next(S, e)$ is s'; (2) if e is a channel directed towards p, then the state of c in $next(S, e)$ is c's state in S with message M deleted from its head; and (3) if c is a channel directed away from p, then the state of c in $next(S, e)$ is the same as c's state in S with message M added to the tail.

Let $seq = (e_i\colon 0 \le i \le n)$ be a sequence of events in component processes of a distributed system. We say that *seq* is a *computation of the system* if and only if event e_i can occur in global state S_i, $0 \le i \le n$, where S_0 is the initial global state and

$$S_{i+1} = next(S_i, e_i) \qquad \text{for} \quad 0 \le i \le n.$$

An alternate model, based on Lamport [6], which views computations as partially ordered sets of events, is given in [7].

Example 2.1. To illustrate the definition of a distributed system, consider a simple system consisting of two processes p and q, and two channels c and c' as shown in Figure 2.

The system contains one *token* that is passed from one process to another, and hence we call this system the "single-token conservation" system. Each process has two states, s_0 and s_1, where s_0 is the state in which the process does not possess the token and s_1 is the state in which it does. The initial state of p is s_1 and of q is s_0. Each process has two events: (1) a transition from s_1 to s_0 with the

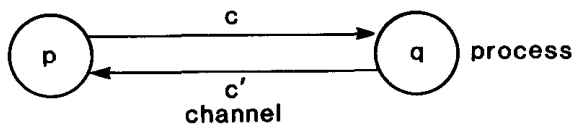


Fig. 2. The simple distributed system of Examples 2.1 and 2.2.

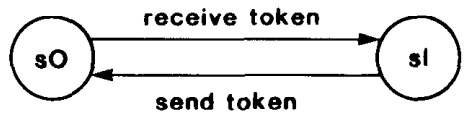


Fig. 3. State-transition diagram of a process in Example 2.1.

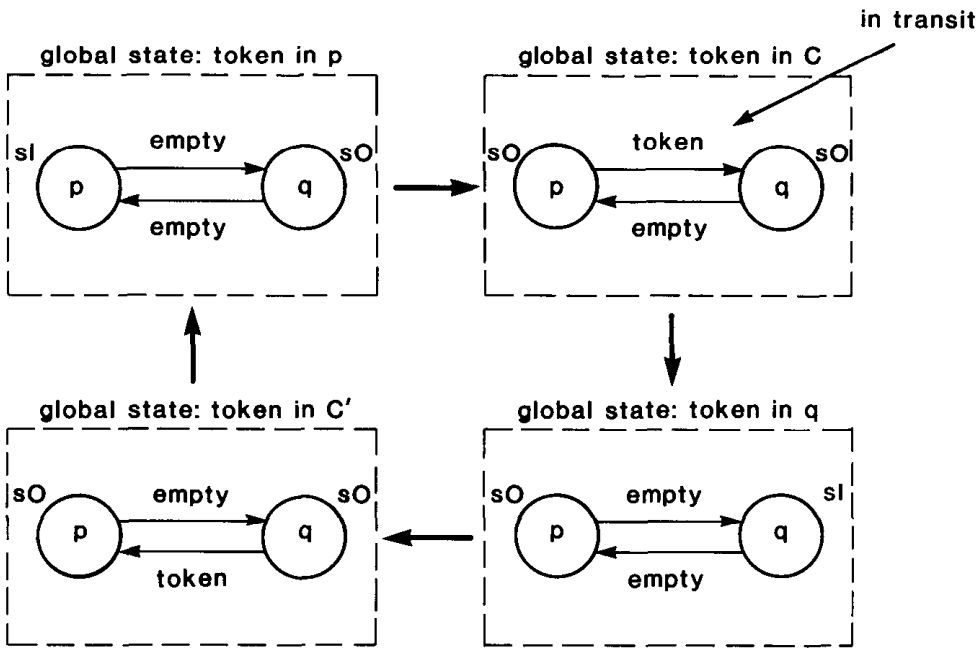


Fig. 4. Global states and transitions of the single-token conservation system.

sending of the token, and (2) a transition from s_0 to s_1 with the receipt of the token. The state-transition diagram for a process is shown in Figure 3. The global states and transitions are shown in Figure 4.

A system computation corresponds to a path in the global-state-transition diagram (Figure 4) starting at the initial global state. Examples of system computations are: (1) the empty sequence and (2) ⟨p sends token, q receives token, q sends token⟩. The following sequence is not a computation of the system: ⟨p sends token, q sends token⟩, because the event "q sends token" cannot occur while q is in the state s_0.

For brevity, the four global states, in order of transition (see Figure 4), will be called (1) in-p, (2) in-c, (3) in-q, and (4) in-c', to denote the location of the token. This example will be used later to motivate the algorithm. □

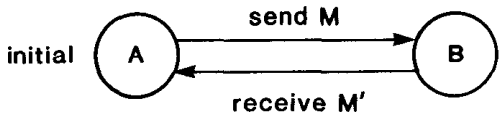


Fig. 5. State-transition diagram for process p in Example 2.2.

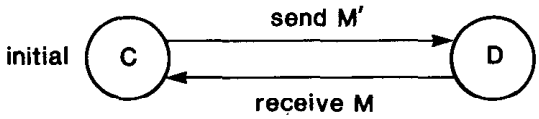


Fig. 6. State-transition diagram for process q in Example 2.2.

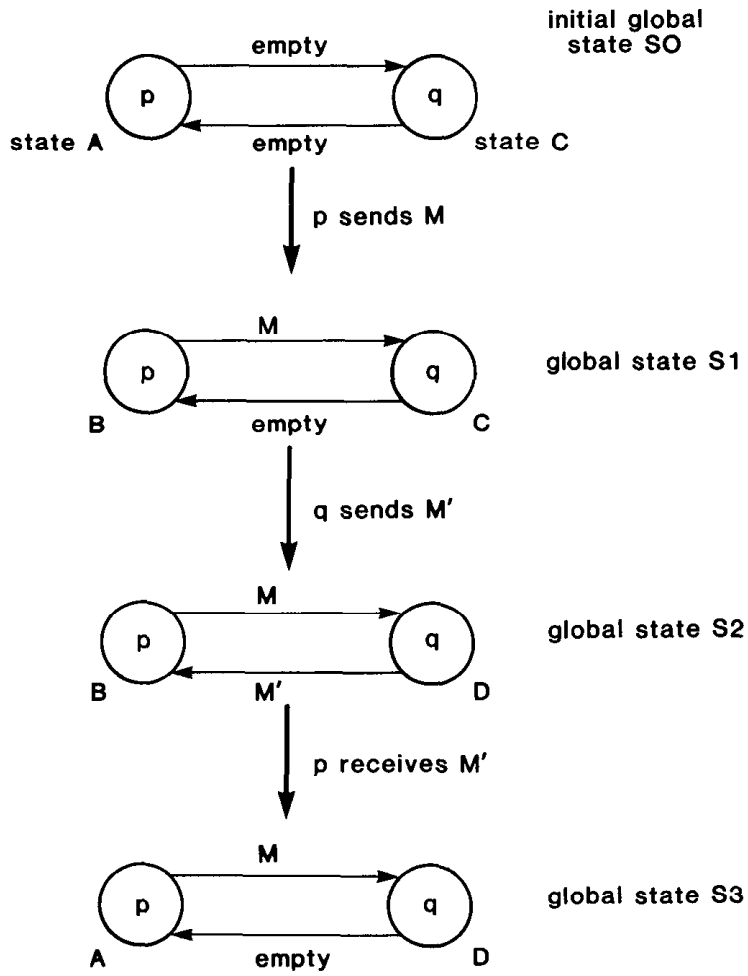


Fig. 7. A computation for Example 2.2.

Example 2.2. This example illustrates nondeterministic computations. Nondeterminism plays an interesting role in the snapshot algorithm.

In Example 2.1 there is exactly one event possible in each global state. Consider a system with the same topology as Example 2.1 (see Figure 2) but where the processes p and q are defined by the state-transition diagrams of Figures 5 and 6.

An example of a computation is shown in Figure 7. The reader should observe that there may be more than one transition allowable from a global state. For instance, events "p sends M" and "q sends M'" may occur in the initial global state, and the next states after these events are different. □

3. THE ALGORITHM

3.1. Motivation for the Steps of the Algorithm

The global-state recording algorithm works as follows: Each process records its own state, and the two processes that a channel is incident on cooperate in recording the channel state. We cannot ensure that the states of all processes and channels will be recorded at the same instant because there is no global clock; however, we require that the recorded process and channel states form a "meaningful" global system state.

The global-state recording algorithm is to be superimposed on the underlying computation, that is, it must run concurrently with, but not alter, the underlying computation. The algorithm may send messages and require processes to carry out computations; however, the messages and computation required to record the global state must not interfere with the underlying computation.

We now consider an example to motivate the steps of the algorithm. In the example we shall assume that we can record the state of a channel instantaneously; we postpone discussion of how the channel state is recorded. Let c be a channel from p to q. The purpose of the example is to gain an intuitive understanding of the relationship between the instant at which the state of channel c is to be recorded and the instants at which the states of processes p and q are to be recorded.

Example 3.1. Consider the single-token conservation system. Assume that the state of process p is recorded in global state in-p. Then the state recorded for p shows the token in p. Now assume that the global state transits to in-c (because p sends the token). Suppose the states of channels c and c' and of process q were recorded in global state in-c, so the state recorded for channel c shows it with the token and the states recorded for channel c' and process q show them not in possession of the token. The composite global state recorded in this fashion would show two tokens in the system, one in p and the other in c. But a global state with two tokens is unreachable from the initial global state in a *single-token* conservation system! The inconsistency arises because the state of p is recorded *before* p sent a message along c and the state of c is recorded *after* p sent the message. Let n be the number of messages sent along c before p's state is recorded, and let n' be the number of messages sent along c before c's state is recorded. Our example suggests that the recorded global state may be inconsistent if $n < n'$.

Now consider an alternate scenario. Suppose the state of c is recorded in global state in-p, the system then transits to global state in-c, and the states of c', p, and q are recorded in global state in-c. The recorded global state shows *no* tokens in the system. This example suggests that the recorded global state may be inconsistent if the state of c is recorded *before* p sends a message along c and the state of p is recorded *after* p sends a message along c, that is, if $n > n'$. □

We learn from these examples that (in general) a consistent global state requires

$$n = n'. \tag{1}$$

Let m be the number of messages received along c before q's state is recorded. Let m' be the number of messages received along c before c's state is recorded. We leave it up to the reader to extend the example to show that consistency requires

$$m = m'. \tag{2}$$

In every state, the number of messages received along a channel cannot exceed the number of messages sent along that channel, that is,

$$n' \geq m'. \tag{3}$$

From the above equations,

$$n \geq m. \tag{4}$$

The state of channel c that is recorded must be the sequence of messages sent along the channel before the sender's state is recorded, excluding the sequence of messages received along the channel before the receiver's state is recorded—that is, if $n' = m'$, the recorded state of c must be the empty sequence, and if $n' > m'$, the recorded state of c must be the $(m' + 1)$st, . . . , n'th messages sent by p along c. This fact and eqs. (1)–(4) suggest a simple algorithm by which q can record the state of channel c. Process p sends a special message, called a *marker*, after the nth message it sends along c (and before sending further messages along c). The marker has no effect on the underlying computation. The state of c is the sequence of messages received by q after q records its own state and before q receives the marker along c. To ensure eq. (4), q must record its state, if it has not done so already, after receiving a marker along c and before q receives further messages along c.

Our example suggests the following outline for a global state detection algorithm.

3.2 Global-State-Detection Algorithm Outline

Marker-Sending Rule for a Process p. For each channel c, incident on, and directed away from p:

p sends one marker along c after p records its state and before p sends further messages along c.

Marker-Receiving Rule for a Process q. On receiving a marker along a channel c:

if q has not recorded its state **then**
  begin q records its state;
    q records the state c as the empty sequence
  end
else q records the state of c as the sequence of messages received along c after q's state was recorded and before q received the marker along c.

3.3 Termination of the Algorithm

The marker receiving and sending rules guarantee that if a marker is received along every channel, then each process will record its state and the states of all

incoming channels. To ensure that the global-state recording algorithm terminates in finite time, each process must ensure that (L1) no marker remains forever in an incident input channel and (L2) it records its state within finite time of initiation of the algorithm.

The algorithm can be initiated by one or more processes, each of which records its state spontaneously, without receiving markers from other processes; we postpone discussion of what may cause a process to record its state spontaneously. If process p records its state and there is a channel from p to a process q, then q will record its state in finite time because p will send a marker along the channel and q will receive the marker in finite time (L1). Hence if p records its state and there is a path (in the graph representing the system) from p to a process q, then q will record its state in finite time because, by induction, every process along the path will record its state in finite time. Termination in finite time is ensured if for every process q: q spontaneously records its state or there is a path from a process p, which spontaneously records its state, to q.

In particular, if the graph is strongly connected and at least one process spontaneously records its state, then all processes will record their states in finite time (provided L1 is ensured).

The algorithm described so far allows each process to record its state and the states of incoming channels. The recorded process and channel states must be collected and assembled to form the recorded global state. We shall not describe algorithms for collecting the recorded information because such algorithms have been described elsewhere [4, 10]. A simple algorithm for collecting information in a system whose topology is strongly connected is for each process to send the information it records along all outgoing channels, and for each process receiving information for the first time to copy it and propagate it along all of its outgoing channels. All the recorded information will then get to all the processes in finite time, allowing all processes to determine the recorded global state.

4. PROPERTIES OF THE RECORDED GLOBAL STATE

To gain an intuitive understanding of the properties of the global state recorded by the algorithm, we shall study Example 2.2. Assume that the state of p is recorded in global state S_0 (Figure 7), so the state recorded for p is A. After recording its state, p sends a marker along channel c. Now assume that the system goes to global state S_1, then S_2, and then S_3 while the marker is still in transit, and the marker is received by q when the system is in global state S_3. On receiving the marker, q records its state, which is D, and records the state of c to be the empty sequence. After recording its state, q sends a marker along channel c'. On receiving the marker, p records the state of c' as the sequence consisting of the single message M'. The recorded global state S^* is shown in Figure 8. The recording algorithm was initiated in global state S_0 and terminated in global state S_3.

Observe that the global state S^* recorded by the algorithm is not identical to any of the global states S_0, S_1, S_2, S_3 that occurred in the computation. Of what use is the algorithm if the recorded global state never occurred? We shall now answer this question.

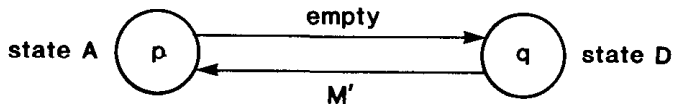


Fig. 8. A recorded global state for Example 2.2.

Let $seq = (e_i, 0 \leq i)$ be a distributed computation, and let S_i be the global state of the system immediately before event e_i, $0 \leq i$, in *seq*. Let the algorithm be initiated in global state S_ι and let it terminate in global state S_ϕ, $0 \leq \iota \leq \phi$; in other words, the algorithm is initiated after $e_{\iota-1}$ if $\iota > 0$, and before e_ι, and it terminates after $e_{\phi-1}$ if $\phi > 0$, and before e_ϕ. We observed in Example 2.2 that the recorded global state S^* may be different from all global states S_k, $\iota \leq k \leq \phi$.

We shall show that:

(1) S^* is reachable from S_ι, and
(2) S_ϕ is reachable from S^*.

Specifically, we shall show that there exists a computation *seq'* where

(1) *seq'* is a permutation of *seq*, such that S_ι, S^* and S_ϕ occur as global states in *seq'*,
(2) $S_\iota = S^*$ or S_ι occurs earlier than S^*, and
(3) $S_\phi = S^*$ or S^* occurs earlier than S_ϕ in *seq'*.

THEOREM 1. *There exists a computation* $seq' = (e_i', 0 \leq i)$ *where*

(1) *For all i, where* $i < \iota$ *or* $i \geq \phi$: $e_i' = e_i$, and
(2) *the subsequence* $(e_i', \iota \leq i < \phi)$ *is a permutation of the subsequence* $(e_i, \iota \leq i < \phi)$, *and*
(3) *for all i where* $i \leq \iota$ *or* $i \geq \phi$: $S_i' = S_i$, *and*
(4) *there exists some k,* $\iota \leq k \leq \phi$, *such that* $S^* = S_k'$.

PROOF. Event e_i in *seq* is called a *prerecording* event if and only if e_i is in a process p and p records its state *after* e_i in *seq*. Event e_i in *seq* is called a *postrecording* event if and only if it is not a prerecording event—that is, if e_i is in a process p and p records its state *before* e_i in *seq*. All events e_i, $i < \iota$, are prerecording events and all events e_i, $i \geq \phi$, are postrecording events in *seq*. There *may be* a postrecording event e_{j-1} before a prerecording event e_j for some j, $\iota < j < \phi$; this can occur only if e_{j-1} and e_j are in different processes (because if e_{j-1} and e_j are in the same process and e_{j-1} is a postrecording event, then so is e_j).

We shall derive a computation *seq'* by permuting *seq*, where all prerecording events occur before all postrecording events in *seq'*. We shall show that S^* is the global state in *seq'* after all prerecording events and before all postrecording events.

Assume that there is a postrecording event e_{j-1} before a prerecording event e_j in *seq*. We shall show that the sequence obtained by interchanging e_{j-1} and e_j must also be a computation. Events e_{j-1} and e_j must be on different processes. Let p be the process in which e_{j-1} occurs, and let q be the process in which e_j occurs. There cannot be a message sent at e_{j-1} which is received at e_j because (1)

if a message is sent along a channel c when event e_{j-1} occurs, then a marker must have been sent along c *before* e_{j-1}, since e_{j-1} is a postrecording event, and (2) if the message is received along channel c when e_j occurs, then the marker must have been received along c *before* e_j occurs (since channels are first-in-first-out), in which case (by the marker-receiving rule) e_j would be a postrecording event too.

The state of process q is not altered by the occurrence of event e_{j-1} because e_{j-1} is in a different process p. If e_j is an event in which q receives a message M along a channel c, then M must have been the message at the head of c before event e_{j-1}, since a message sent at e_{j-1} cannot be received at e_j. Hence event e_j can occur in global state S_{j-1}.

The state of process p is not altered by the occurrence of e_j. Hence e_{j-1} can occur after e_j. Hence the sequence of events $e_1, \ldots, e_{j-2}, e_j, e_{j-1}$ is a computation. From the arguments in the last paragraph it follows that the global state after computation $e_1, \ldots, e_j$ is the same as the global state after computation $e_1, \ldots, e_{j-2}, e_j, e_{j-1}$.

Let seq^* be a permutation of seq that is identical to seq except that e_j and e_{j-1} are interchanged. Then seq^* must also be a computation. Let $\bar{S}_i$ be the global state immediately before the ith event in seq^*. From the arguments of the previous paragraph,

$$\bar{S}_i = S_i \qquad \text{for all} \quad i \quad \text{where} \quad i \neq j.$$

By repeatedly swapping postrecording events that immediately follow prerecording events, we see that there exists a permutation seq' of seq in which

(1) all prerecording events precede all postrecording events,
(2) seq' is a computation,
(3) for all i where $i < \iota$ or $i \geq \phi$: $e_i' = e_i$, and
(4) for all i where $i \leq \iota$ or $i \geq \phi$: $S_i' = S_i$.

Now we shall show that the global state after all prerecording events and before all postrecording events in seq' is S^*. To do this, we need to show that

(1) the state of each process p in S^* is the same as its state after the process computation consisting of the sequence of prerecorded events on p, and
(2) the state of each channel c in S^* is (sequence of messages corresponding to prerecorded sends on c) − (sequence of messages corresponding to prerecorded receives on c).

The proof of the first part is trivial. Now we prove part (2). Let c be a channel from process p to process q. The state of channel c recorded in S^* is the sequence of messages received on c by q *after* q records its state *and before* q receives a marker on c. The sequence of messages sent by p along c before p sends a marker along c is the sequence corresponding to prerecorded sends on c. Part (2) now follows. □

Example 4.1. The purpose of this example is to show how the computation seq' is derived from the computation seq. Consider Example 2.2. The sequence

of events shown in the computation of Figure 7 is

e_0: p sends M and changes state to B (a postrecording event)

e_1: q sends M' and changes state to D (a prerecording event)

e_2: p receives M' and changes state to A (a postrecording event)

Since e_0, a postrecording event, immediately precedes e_1, a prerecording event, we interchange them, to get the permuted sequence seq':

e'_0: q sends M' and changes state to D (a prerecording event)

e'_1: p sends M and changes state to B (a postrecording event)

e'_2: p receives M' and changes state to A (a postrecording event)

In seq', all prerecording events precede all postrecording events. We leave it to the reader to show that the global state after e'_0 is the recorded global state.

5. STABILITY DETECTION

We now solve the stability-detection problem described in Section 1. We study the stability-detection problem because it is a paradigm for many practical problems, such as distributed deadlock detection.

A stability-detection algorithm is defined as follows:

Input: A stable property y

Output: A Boolean value *definite* with the property:
$(y(S_\iota) \rightarrow definite)$ and $(definite \rightarrow y(S_\phi))$

where S_ι and S_ϕ are the global states of the system when the algorithm is initiated and when it terminates, respectively. (The symbol $\rightarrow$ denotes logical implication.)

The input to the algorithm is (the definition of) function y. During the execution of the algorithm the value $y(S)$ for some global state S may be determined by a process in the system by applying the *externally defined* function y to global state S. By the output of the algorithm being a Boolean value *definite* we mean that (1) some specially designated process (say p) enters and thereafter remains in some special state to symbolize an output of *definite* = *true*, and (2) p enters and remains in some other special state to symbolize an output of *definite* = *false*.

Definite = *true* implies that the stable property holds when the algorithm *terminates*. However, *definite* = *false* implies that the stable property does not hold when the algorithm is *initiated*. We emphasize that *definite* = *true* gives us information about the state of the system at the *termination* of the algorithm, whereas *definite* = *false* gives us information about the system state at the *initiation* of the algorithm. In particular, we cannot deduce from *definite* = *false* that the stable property does not hold at termination of the algorithm.

The solution to the stability detection problem is

```
begin
    record a global state S*;
    definite := y(S*)
end.
```

The correctness of the stability detection algorithm follows from the following facts:

(1) S^* is reachable from S_ι,
(2) S_ϕ is reachable from S^* (Theorem 1), and
(3) $y(S) \rightarrow y(S')$ for all S' reachable from S (definition of a stable property).

ACKNOWLEDGMENTS

J. Misra's contributions in defining the problem of global state detection are gratefully acknowledged. We are grateful to E. W. Dijkstra and C. S. Scholten for their comments—particularly regarding the proof of Theorem 1. The outline of the current version of the proof was suggested by them. Dijkstra's note [3] on the subject provides colorful insight into the problem of stability detection. Thanks are due to C. A. R. Hoare, F. Schneider, and G. Andrews who helped us with detailed comments. We are grateful to Anita Jones and anonymous referees for suggestions.

REFERENCES

1. CHANDY, K. M., AND MISRA, J. Distributed computation on graphs: Shortest path algorithms. *Commun. ACM 25,* 11 (Nov. 1982), 833–837.
2. CHANDY, K. M., MISRA, J., AND HAAS, L. Distributed deadlock detection. *ACM Trans. Comput. Syst.* 1, 2 (May 1983), 144–156.
3. DIJKSTRA, E. W. The distributed snapshot of K. M. Chandy and L. Lamport. Tech. Rep. EWD 864a, Univ. of Texas, Austin, Tex., 1984.
4. DIJKSTRA, E. W., AND SCHOLTEN, C. S. Termination detection for diffusing computations. *Inf. Proc. Lett. 11,* 1 (Aug. 1980), 1–4.
5. GLIGOR, V. D., AND SHATTUCK, S. H. Deadlock detection in distributed systems. *IEEE Trans. Softw. Eng. SE-6,* 5 (Sep. 1980), 435–440.
6. LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM 21,* 7 (Jul. 1978), 558–565.
7. LAMPORT, L., AND CHANDY, K. M. On partially-ordered event models of distributed computations. Submitted for publication.
8. MAHOUD, S. A., AND RIORDAN, J. S. Software controlled access to distributed databases. *INFOR 15,* 1 (Feb. 1977), 22–36.
9. MENASCE, D., AND MUNTZ, R. Locking and deadlock detection in distributed data bases. *IEEE Trans. Softw. Eng. SE-5,* 3 (May 1979), 195–202.
10. MISRA, J., AND CHANDY, K. M. Termination detection of diffusing computations in communicating sequential processes. *ACM Trans. Program. Lang. Syst.* 4, 1 (Jan. 1982), 37–43.
11. OBERMARCK, R. Distributed deadlock detection algorithm. *ACM Trans. Database Syst. 7,* 2 (Jun. 1982), 187–208.

Received January 1984; revised September 1984; accepted 7 December 1984

Chapter 3

Impossibility of Distributed Consensus with One Faulty Process

Authors:
Michael J. Fischer, Nancy A. Lynch, Michael S. Paterson

Publication year: 1985

Impossibility of Distributed Consensus with One Faulty Process

MICHAEL J. FISCHER

Yale University, New Haven, Connecticut

NANCY A. LYNCH

Massachusetts Institute of Technology, Cambridge, Massachusetts

AND

MICHAEL S. PATERSON

University of Warwick, Coventry, England

Abstract. The consensus problem involves an asynchronous system of processes, some of which may be unreliable. The problem is for the reliable processes to agree on a binary value. In this paper, it is shown that every protocol for this problem has the possibility of nontermination, even with only one faulty process. By way of contrast, solutions are known for the synchronous case, the "Byzantine Generals" problem.

Categories and Subject Descriptors: C.2.2 **[Computer-Communication Networks]**: Network Protocols–*protocol architecture*; C.2.4 **[Computer-Communication Networks]**: Distributed Systems–*distributed applications; distributed databases; network operating systems*; C.4 **[Performance of Systems]**: Reliability, Availability, and Serviceability; F.1.2 **[Computation by Abstract Devices]**: Modes of Computation–*parallelism*; H.2.4 **[Database Management]**: Systems–*distributed systems*; *transaction processing*

General Terms: Algorithms, Reliability, Theory

Additional Key Words and Phrases: Agreement problem, asynchronous system, Byzantine Generals problem, commit problem, consensus problem, distributed computing, fault tolerance, impossibility proof, reliability

1. *Introduction*

The problem of reaching agreement among remote processes is one of the most fundamental problems in distributed computing and is at the core of many

Editing of this paper was performed by guest editor S. L. Graham. The Editor-in-Chief of JACM did not participate in the processing of the paper.

This work was supported in part by the Office of Naval Research under Contract N00014-82-K-0154, by the Office of Army Research under Contract DAAG29-79-C-0155, and by the National Science Foundation under Grants MCS-7924370 and MCS-8116678.

This work was originally presented at the 2nd ACM Symposium on Principles of Database Systems, March 1983.

Authors' present addresses: M. J. Fischer, Department of Computer Science, Yale University, P.O. Box 2158, Yale Station, New Haven, CT 06520; N. A. Lynch, Laboratory for Computer Science, Massachusetts Institute of Technology, 545 Technology Square, Cambridge, MA 02139; M. S. Paterson, Department of Computer Science, University of Warwick, Coventry CV4 7AL, England

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1985 ACM 0004-5411/85/0400-0374 $00.75

algorithms for distributed data processing, distributed file management, and fault-tolerant distributed applications.

A well-known form of the problem is the "transaction commit problem," which arises in distributed database systems [6, 13, 15–17, 21–24] (see also G. LeLann, private communication, quoted in [15]). The problem is for all the data manager processes that have participated in the processing of a particular transaction to agree on whether to install the transaction's results in the database or to discard them. The latter action might be necessary, for example, if some data managers were, for any reason, unable to carry out the required transaction processing. Whatever decision is made, all data managers must make the same decision in order to preserve the consistency of the database.

Reaching the type of agreement needed for the "commit" problem is straightforward if the participating processes and the network are completely reliable. However, real systems are subject to a number of possible faults, such as process crashes, network partitioning, and lost, distorted, or duplicated messages. One can even consider more Byzantine types of failure [5, 7, 8, 11, 14, 18, 19] in which faulty processes might go completely haywire, perhaps even sending messages according to some malevolent plan. One therefore wants an agreement protocol that is as reliable as possible in the presence of such faults. Of course, any protocol can be overwhelmed by faults that are too frequent or too severe, so the best that one can hope for is a protocol that is tolerant to a prescribed number of "expected" faults.

In this paper, we show the surprising result that no completely asynchronous consensus protocol can tolerate even a single unannounced process death. We do not consider Byzantine failures, and we assume that the message system is reliable—it delivers all messages correctly and exactly once. Nevertheless, even with these assumptions, the stopping of a single process at an inopportune time can cause any distributed commit protocol to fail to reach agreement. Thus, this important problem has no robust solution without further assumptions about the computing environment or still greater restrictions on the kind of failures to be tolerated!

Crucial to our proof is that processing is completely asynchronous; that is, we make no assumptions about the relative speeds of processes or about the delay time in delivering a message. We also assume that processes do not have access to synchronized clocks, so algorithms based on time-outs, for example, cannot be used. (In particular, the solutions in [6] are not applicable.) Finally, we do not postulate the ability to detect the death of a process, so it is impossible for one process to tell whether another has died (stopped entirely) or is just running very slowly.

Our impossibility result applies to even a very weak form of the *consensus problem*. Assume that every process starts with an initial value in $\{0, 1\}$. A nonfaulty process decides on a value in $\{0, 1\}$ by entering an appropriate decision state. All nonfaulty processes that make a decision are required to choose the same value. For the purpose of the impossibility proof, we require only that *some* process eventually make a decision. (Of course, any algorithm of interest would require that all nonfaulty processes make a decision.) The trivial solution in which, say, 0 is always chosen is ruled out by stipulating that both 0 and 1 are possible decision values, although perhaps for different initial configurations.

Our system model is rather strong so as to make our impossibility proof as widely applicable as possible. Processes are modeled as automata (with possibly infinitely many states) that communicate by means of messages. In one atomic step, a process can attempt to receive a message, perform local computation on the basis of

whether or not a message was delivered to it (and if so, which one), and send an arbitrary but finite set of messages to other processes. In particular, an "atomic broadcast" capability is assumed, so a process can send the same message in one step to all other processes with the knowledge that if any nonfaulty process receives the message, then all the nonfaulty processes will. Every message is eventually delivered as long as the destination process makes infinitely many attempts to receive, but messages can be delayed, arbitrarily long, and delivered out of order.

The asynchronous commit protocols in current use all seem to have a "window of vulnerability"—an interval of time during the execution of the algorithm in which the delay or inaccessibility of a single process can cause the entire algorithm to wait indefinitely. It follows from our impossibility result that every commit protocol has such a "window," confirming a widely believed tenet in the folklore.

2. *Consensus Protocols*

A *consensus protocol* P is an asynchronous system of N processes ($N \geq 2$). Each process p has a one-bit *input register* x_p, an *output register* y_p with values in $\{b, 0, 1\}$, and an unbounded amount of internal storage. The values in the input and output registers, together with the program counter and internal storage, comprise the *internal state. Initial states* prescribe fixed starting values for all but the input register; in particular, the output register starts with value b. The states in which the output register has value 0 or 1 are distinguished as being *decision states.* p acts deterministically according to a *transition* function. The transition function cannot change the value of the output register once the process has reached a decision state; that is, the output register is "write-once." The entire system P is specified by the transition functions associated with each of the processes and the initial values of the input registers.

Processes communicate by sending each other messages. A *message* is a pair (p, m), where p is the name of the destination process and m is a "message value" from a fixed universe M. The *message system* maintains a multiset, called the *message buffer*, of messages that have been sent but not yet delivered. It supports two abstract operations:

send(p, m): Places (p, m) in the message buffer;
receive(p): Deletes some message (p, m) from the buffer and returns m, in which case we say (p, m) is *delivered*, or returns the special null marker $\varnothing$ and leaves the buffer unchanged.

Thus, the message system acts nondeterministically, subject only to the condition that if receive(p) is performed infinitely many times, then every message (p, m) in the message buffer is eventually delivered. In particular, the message system is allowed to return $\varnothing$ a finite number of times in response to receive(p), even though a message (p, m) is present in the buffer.

A *configuration* of the system consists of the internal state of each process, together with the contents of the message buffer. An *initial configuration* is one in which each process starts at an initial state and the message buffer is empty.

A *step* takes one configuration to another and consists of a primitive step by a single process p. Let C be a configuration. The step occurs in two phases. First, receive(p) is performed on the message buffer in C to obtain a value $m \in M \cup \{\varnothing\}$. Then, depending on p's internal state in C and on m, p enters a new internal state and sends a finite set of messages to other processes. Since processes are deterministic, the step is completely determined by the pair $e = (p, m)$, which we

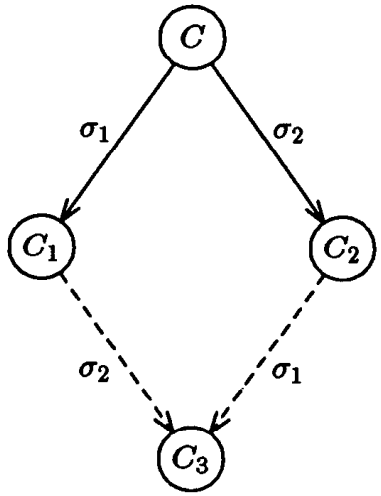


FIGURE 1

call an *event*. (This "event" should be thought of as the receipt of m by p.) $e(C)$ denotes the resulting configuration, and we say that e can be *applied* to C. Note that the event $(p, \varnothing)$ can always be applied to C, so it is always possible for a process to take another step.

A *schedule* from C is a finite or infinite sequence σ of events that can be applied, in turn, starting from C. The associated sequence of steps is called a *run*. If σ is finite, we let $\sigma(C)$ denote the resulting configuration, which is said to be *reachable* from C. A configuration reachable from some initial configuration is said to be *accessible*. Hereafter, all configurations mentioned are assumed to be accessible.

The following lemma expresses a "commutativity" property of schedules.

LEMMA 1. *Suppose that from some configuration C, the schedules σ_1, σ_2 lead to configurations C_1, C_2, respectively. If the sets of processes taking steps in σ_1 and σ_2, respectively, are disjoint, then σ_2 can be applied to C_1 and σ_1 can be applied to C_2, and both lead to the same configuration C_3.* (See Figure 1.)

PROOF. The result follows at once from the system definition, since σ_1 and σ_2 do not interact. □

A configuration C has *decision value* v if some process p is in a decision state with $y_p = v$. A consensus protocol is *partially correct* if it satisfies two conditions:

(1) No accessible configuration has more than one decision value.
(2) For each $v \in \{0, 1\}$, some accessible configuration has decision value v.

A process p is *nonfaulty* in a run provided that it takes infinitely many steps, and it is *faulty* otherwise. A run is *admissible* provided that at most one process is faulty and that all messages sent to nonfaulty processes are eventually received.

A run is a *deciding* run provided that some process reaches a decision state in that run. A consensus protocol P is *totally correct in spite of one fault* if it is partially correct, and every admissible run is a deciding run. Our main theorem shows that every partially correct protocol for the consensus problem has some admissible run that is not a deciding run.

3. *Main Result*

THEOREM 1. *No consensus protocol is totally correct in spite of one fault.*

PROOF. Assume to the contrary that P is a consensus protocol that is totally correct in spite of one fault. We prove a sequence of lemmas which eventually lead to a contradiction.

The basic idea is to show circumstances under which the protocol remains forever indecisive. This involves two steps. First, we argue that there is some initial configuration in which the decision is not already predetermined. Second, we construct an admissible run that avoids ever taking a step that would commit the system to a particular decision.

Let C be a configuration and let V be the set of decision values of configurations reachable from C. C is *bivalent* if $|V| = 2$. C is *univalent* if $|V| = 1$, let us say 0-*valent* or 1-*valent* according to the corresponding decision value. By the total correctness of P, and the fact that there are always admissible runs, $V \neq \varnothing$. □

LEMMA 2. *P has a bivalent initial configuration.*

PROOF. Assume not. Then P must have both 0-valent and 1-valent initial configurations by the assumed partial correctness. Let us call two initial configurations *adjacent* if they differ only in the initial value x_p of a single process p. Any two initial configurations are joined by a chain of initial configurations, each adjacent to the next. Hence, there must exist a 0-valent initial configuration C_0 adjacent to a 1-valent initial configuration C_1. Let p be the process in whose initial value they differ.

Now consider some admissible deciding run from C_0 in which process p takes no steps, and let σ be the associated schedule. Then σ can be applied to C_1 also, and corresponding configurations in the two runs are identical except for the internal state of process p. It is easily shown that both runs eventually reach the same decision value. If the value is 1, then C_0 is bivalent; otherwise, C_1 is bivalent. Either case contradicts the assumed nonexistence of a bivalent initial configuration. □

LEMMA 3. *Let C be a bivalent configuration of P, and let $e = (p, m)$ be an event that is applicable to C. Let $\mathscr{C}$ be the set of configurations reachable from C without applying e, and let $\mathscr{D} = e(\mathscr{C}) = \{e(E) \mid E \in \mathscr{C}$ and e is applicable to $E\}$. Then, $\mathscr{D}$ contains a bivalent configuration.*

PROOF. Since e is applicable to C, then by definition of $\mathscr{C}$ and the fact that messages can be delayed arbitrarily, e is applicable to every $E \in \mathscr{C}$.

Now assume that $\mathscr{D}$ contains no bivalent configurations, so every configuration $D \in \mathscr{D}$ is univalent. We proceed to derive a contradiction.

Let E_i be an i-valent configuration reachable from C, $i = 0, 1$. (E_i exists since C is bivalent.) If $E_i \in \mathscr{C}$, let $F_i = e(E_i) \in \mathscr{D}$. Otherwise, e was applied in reaching E_i, and so there exists $F_i \in \mathscr{D}$ from which E_i is reachable. In either case, F_i is i-valent since F_i is not bivalent (since $F_i \in \mathscr{D}$ and $\mathscr{D}$ contains no bivalent configurations) and one of E_i and F_i is reachable from the other. Since $F_i \in \mathscr{D}$, $i = 0, 1$, $\mathscr{D}$ contains both 0-valent and 1-valent configurations.

Call two configurations *neighbors* if one results from the other in a single step. By an easy induction, there exist neighbors C_0, $C_1 \in \mathscr{C}$ such that $D_i = e(C_i)$ is i-valent, $i = 0, 1$. Without loss of generality, $C_1 = e'(C_0)$ where $e' = (p', m')$.

Case 1. If $p' \neq p$, then $D_1 = e'(D_0)$ by Lemma 1. This is impossible, since any successor of a 0-valent configuration is 0-valent. (See Figure 2.)

Case 2. If $p' = p$, then consider any finite deciding run from C_0 in which p takes no steps.

Let σ be the corresponding schedule, and let $A = \sigma(C_0)$. By Lemma 1, σ is applicable to D_i, and it leads to an i-valent configuration $E_i = \sigma(D_i)$, $i = 0, 1$. Also

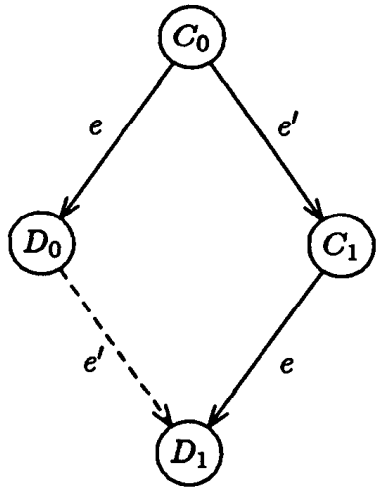


FIGURE 2

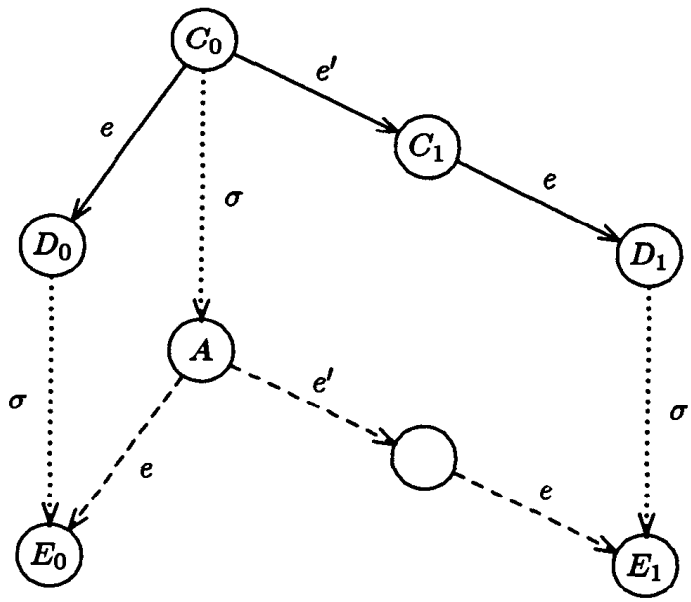


FIGURE 3

by Lemma 1, $e(A) = E_0$ and $e(e'(A)) = E_1$. (See Figure 3.) Hence, A is bivalent. But this is impossible since the run to A is deciding (by assumption), so A must be univalent.

In each case, we reached a contradiction, so $\mathscr{D}$ contains a bivalent configuration. □

Any deciding run from a bivalent initial configuration goes to a univalent configuration, so there must be some single step that goes from a bivalent to a univalent configuration. Such a step determines the eventual decision value. We now show that it is always possible to run the system in a way that avoids such steps, leading to an admissible nondeciding run.

The run is constructed in stages, starting from an initial configuration. We ensure that the run is admissible in the following way. A queue of processes is maintained, initially in an arbitrary order, and the message buffer in a configuration is ordered according to the time the messages were sent, earliest first. Each stage consists of one or more process steps. The stage ends with the first process in the process queue taking a step in which, if its message queue was not empty at the start of the

stage, its earliest message is received. This process is then moved to the back of the process queue. In any infinite sequence of such stages every process takes infinitely many steps and receives every message sent to it. The run is therefore admissible. Our problem, of course, is to do this in such a way as to avoid a decision ever being reached.

Let C_0 be a bivalent initial configuration whose existence is assured by Lemma 2. Execution begins in C_0, and we ensure that every stage begins from a bivalent configuration. Suppose then that configuration C is bivalent and that process p heads the priority queue. Let m be the earliest message to p in C's message buffer, if any, and $\varnothing$ otherwise. Let $e = (p, m)$. By Lemma 3, there is a bivalent configuration C' reachable from C by a schedule in which e is the last event applied. The corresponding sequence of steps defines the stage.

Since each stage ends in a bivalent configuration, every stage in the construction of the infinite schedule succeeds. The resulting run is admissible, and no decision is ever reached. It follows that P is not totally correct. □

4. *Initially Dead Processes*

In this section, we exhibit a protocol that solves the consensus problem for N processes as long as a majority of the processes are nonfaulty and no process dies during the execution of the protocol. No process knows in advance, however, which of the processes are initially dead and which are not.

The protocol works in two stages. During the first stage, the processes construct a directed graph G with a node corresponding to each process. Every process broadcasts a message containing its process number and then listens for messages from $L - 1$ other processes, where $L = \lceil (N + 1)/2 \rceil$. G has an edge from i to j iff j receives a message from i. Thus, G has indegree $L - 1$.

In the second stage, the processes construct G^+ (the transitive closure of G) in the sense that upon completion of this stage, each process k knows about all of the edges (j, k) incident on k in G^+ as well as the initial values of all such j.

To carry out this stage, each process broadcasts to all other processes its process number and initial value together with the names of the $L - 1$ processes it heard from during the first stage. It then waits until it has received a stage 2 message from every ancestor in G that it knows about. Initially, it knows only about the $L - 1$ processes from which it heard directly during the first stage, but it learns about additional ancestors from the stage 2 messages that it receives. Waiting continues until such time as all currently known-about processes have been heard from.

At this point, each process knows all of its own ancestors and the edges of G incident on them. Using this information, it computes all of the edges of G^+ incident on each of its ancestors. It then determines which of its ancestors belong to an *initial* clique of G^+, that is, a clique with no incoming edges. To do this, it uses the fact that a node k is in an initial clique iff k is itself an ancestor of every node j that is an ancestor of k. Since every node in G^+ has at least $L - 1$ predecessors, there can be only one initial clique; it has cardinality at least L, and every process that completes the second stage knows exactly the set of processes comprising it.

Finally, each process makes a decision based on the initial values of the processes in the initial clique using any agreed-upon rule. Since all processes know the initial values of all members of the initial clique, they all reach the same decision.

The correctness of this protocol proves the following theorem.

THEOREM 2. *There is a partially correct consensus protocol in which all nonfaulty processes always reach a decision, provided no processes die during its execution and a strict majority of the processes are alive initially.*

5. *Conclusion*

We have shown that a natural and important problem of fault-tolerant cooperative computing cannot be solved in a totally asynchronous model of computation. These results do not show that such problems cannot be "solved" in practice; rather, they point up the need for more refined models of distributed computing that better reflect realistic assumptions about processor and communication timings, and for less stringent requirements on the solution to such problems. (For example, termination might be required only with probability 1.) Subsequent to the original announcement of these results [12], progress has been made along both of these lines [1–4, 9, 10, 20, 25].

ACKNOWLEDGMENT. The authors would like to thank John Guttag for helpful discussions during the initial phase of this work, and Gene Stark for discussion of the results and a careful reading of the text. They also thank the referees for pointing out several places where the presentation needed improvement.

REFERENCES

1. ATTIYA, C., DOLEV, D., AND GIL, J. Asynchronous Byzantine consensus. In *Proceedings of the 3rd Annual ACM Symposium on Principles of Distributed Computing* (Vancouver, B.C., Canada, Aug. 27–29). ACM, New York, 1984, pp. 119–133.
2. BEN-OR, M. Another advantage of free choice: Completely asynchronous agreement protocols. In *Proceedings of the 2nd Annual ACM Symposium on Principles of Distributed Computing* (Montreal, Quebec, Canada, Aug. 17–19). ACM, New York, 1983, pp. 27–30.
3. BRACHA, G. An asynchronous $\lfloor(n-1)/3\rfloor$-resilient consensus protocol. In *Proceedings of the 3rd Annual ACM Symposium on Principles of Distributed Computing* (Vancouver, B.C., Canada, Aug. 27–29). ACM, New York, 1984, pp. 154–162.
4. BRACHA, G., AND TOUEG, S. Resilient consensus protocols. In *Proceedings of the 2nd Annual ACM Symposium on Principles of Distributed Computing* (Montreal, Quebec, Canada, Aug. 17–19). ACM, New York, 1983, pp. 12–26.
5. DEMILLO, R. A., LYNCH, N. A., AND MERRITT, M. J. Cryptographic protocols. In *Proceedings of the 14th Annual ACM Symposium on Theory of Computing* (San Francisco, Calif., May 5–7). ACM, New York, 1982, pp. 383–400.
6. DOLEV, D., AND STRONG, H. R. Distributed commit with bounded waiting. In *Proceedings of the 2nd Annual IEEE Symposium on Reliability in Distributed Software and Database Systems.* IEEE, New York, 1982, pp. 53–60.
7. DOLEV, D., AND STRONG, H. R. Polynomial algorithms for multiple processor agreement. In *Proceedings of the 14th Annual ACM Symposium on Theory of Computing* (San Francisco, Calif., May 5–7). ACM, New York, 1982, pp. 401–407.
8. DOLEV, D., FISCHER, M., FOWLER, R., LYNCH, N., AND STRONG, H. R. An efficient algorithm for Byzantine agreement without authentication. *Inf. Control 52*, 3 (1983), 257–274.
9. DOLEV, D., LYNCH, N., PINTER, S., STARK, E., AND WEIHL, W. Reaching approximate agreement in the presence of faults. In *Proceedings of the 3rd Annual IEEE Symposium on Reliability in Distributed Software and Database Systems.* IEEE, New York, 1983, pp. 145–154.
10. DWORK, C., LYNCH, N., AND STOCKMEYER, L. Consensus in the presence of partial synchrony. In *Proceedings of the 3rd Annual ACM Symposium on Principles of Distributed Computing* (Vancouver, B.C., Canada, Aug. 27–29). ACM, New York, 1984, pp. 103–118.
11. FISCHER, M., AND LYNCH. N. A lower bound for the time to assure interactive consistency. *Inf. Proc. Lett. 14*, 4 (1982), 183–186.
12. FISCHER, M., LYNCH, N., AND PATERSON, M. Impossibility of distributed consensus with one faulty process. In *Proceedings of the 2nd Annual ACM SIGACT-SIGMOD Symposium on Principles of Database Systems* (Atlanta, Ga., Mar. 21–23). ACM, New York, 1983, pp. 1–7.

13. GARCIA-MOLINA, H. Elections in a distributed computing system. *IEEE Trans. Comput. C-31*, 1 (1982), 48–59.
14. LAMPORT, L., SHOSTAK, R., AND PEASE, M. The Byzantine Generals problem. *ACM Trans. Prog. Lang. Syst. 4*, 3 (July 1982), 382–401.
15. LAMPSON, B. Replicated Commit. CSL Notebook Entry, Xerox Palo Alto Research Center, Palo Alto, Calif., 1981.
16. LAMPSON, B., AND STURGIS, H. Crash recovery in a distributed data storage system. Manuscript, Xerox Palo Alto Research Center, Palo Alto, Calif., 1979.
17. LINDSAY, B. G., SELINGER, P. G., GALTIERI, C., GRAY, J. N., LORIE, R. A., PRICE, T. G., PUTZOLU, F., TRAIGER, I. L., AND WADE, B. W. Notes on distributed databases. IBM Res. Rep. RJ2571, IBM Research Division, San Jose, Calif., 1979.
18. LYNCH, N., FISCHER, M., AND FOWLER, R. A simple and efficient Byzantine Generals algorithm. In *Proceedings of the 2nd Annual IEEE Symposium on Reliability in Distributed Software and Database Systems*. IEEE, New York, 1982, pp. 46–52.
19. PEASE, M., SHOSTAK, R., AND LAMPORT, L. Reaching agreement in the presence of faults. *J. ACM 27*, 2 (Apr. 1980), 228–234.
20. RABIN, M. Randomized Byzantine Generals. In *Proceedings of the 24th Annual IEEE Symposium on Foundations of Computer Science*. IEEE, New York, 1983, pp. 403–409.
21. REED, D. Naming and synchronization in a decentralized computer system. Ph.D. dissertation, Technical Report MIT/LCS/TR-205, Massachusetts Institute of Technology, Cambridge, Mass., 1978.
22. ROSENKRANTZ, D. J., STEARNS, R. E., AND LEWIS, P. M., II. System level concurrency control for distributed database systems. *ACM Trans. Database Syst. 3*, 2 (June 1978), 178–198.
23. SKEEN, D. A decentralized termination protocol. In *Proceedings of the 2nd Annual IEEE Symposium on Reliability in Distributed Software and Database Systems*. IEEE, New York, 1982, pp. 27–32.
24. SKEEN, D., AND STONEBRAKER, M. A formal model of crash recovery in a distributed system. *IEEE Trans. Softw. Engineering SE-9*, 3 (May 1983), 219–228.
25. TOUEG, S. Randomized Byzantine Agreements. In *Proceedings of the 3rd Annual ACM Symposium on Principles of Distributed Computing* (Vancouver, B.C., Canada, Aug. 27–29). ACM, New York, 1984, pp. 163–178.

RECEIVED SEPTEMBER 1983; REVISED OCTOBER 1984; ACCEPTED OCTOBER 1984

Chapter 4

The Byzantine Generals Problem

Authors:
Leslie Lamport, Robert Shostak, Marshall Pease

Publication year: 1982

The Byzantine Generals Problem

LESLIE LAMPORT, ROBERT SHOSTAK, and MARSHALL PEASE
SRI International

Reliable computer systems must handle malfunctioning components that give conflicting information to different parts of the system. This situation can be expressed abstractly in terms of a group of generals of the Byzantine army camped with their troops around an enemy city. Communicating only by messenger, the generals must agree upon a common battle plan. However, one or more of them may be traitors who will try to confuse the others. The problem is to find an algorithm to ensure that the loyal generals will reach agreement. It is shown that, using only oral messages, this problem is solvable if and only if more than two-thirds of the generals are loyal; so a single traitor can confound two loyal generals. With unforgeable written messages, the problem is solvable for any number of generals and possible traitors. Applications of the solutions to reliable computer systems are then discussed.

Categories and Subject Descriptors: C.2.4. [**Computer-Communication Networks**]: Distributed Systems—*network operating systems*; D.4.4 [**Operating Systems**]: Communications Management—*network communication*; D.4.5 [**Operating Systems**]: Reliability—*fault tolerance*

General Terms: Algorithms, Reliability

Additional Key Words and Phrases: Interactive consistency

1. INTRODUCTION

A reliable computer system must be able to cope with the failure of one or more of its components. A failed component may exhibit a type of behavior that is often overlooked—namely, sending conflicting information to different parts of the system. The problem of coping with this type of failure is expressed abstractly as the Byzantine Generals Problem. We devote the major part of the paper to a discussion of this abstract problem and conclude by indicating how our solutions can be used in implementing a reliable computer system.

We imagine that several divisions of the Byzantine army are camped outside an enemy city, each division commanded by its own general. The generals can communicate with one another only by messenger. After observing the enemy, they must decide upon a common plan of action. However, some of the generals

This research was supported in part by the National Aeronautics and Space Administration under contract NAS1-15428 Mod. 3, the Ballistic Missile Defense Systems Command under contract DASG60-78-C-0046, and the Army Research Office under contract DAAG29-79-C-0102.
Authors' address: Computer Science Laboratory, SRI International, 333 Ravenswood Avenue, Menlo Park, CA 94025.
Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
© 1982 ACM 0164-0925/82/0700-0382 $00.75

may be traitors, trying to prevent the loyal generals from reaching agreement. The generals must have an algorithm to guarantee that

A. All loyal generals decide upon the same plan of action.

The loyal generals will all do what the algorithm says they should, but the traitors may do anything they wish. The algorithm must guarantee condition A regardless of what the traitors do.

The loyal generals should not only reach agreement, but should agree upon a reasonable plan. We therefore also want to insure that

B. A small number of traitors cannot cause the loyal generals to adopt a bad plan.

Condition B is hard to formalize, since it requires saying precisely what a bad plan is, and we do not attempt to do so. Instead, we consider how the generals reach a decision. Each general observes the enemy and communicates his observations to the others. Let $v(i)$ be the information communicated by the ith general. Each general uses some method for combining the values $v(1), \ldots, v(n)$ into a single plan of action, where n is the number of generals. Condition A is achieved by having all generals use the same method for combining the information, and Condition B is achieved by using a robust method. For example, if the only decision to be made is whether to attack or retreat, then $v(i)$ can be General i's opinion of which option is best, and the final decision can be based upon a majority vote among them. A small number of traitors can affect the decision only if the loyal generals were almost equally divided between the two possibilities, in which case neither decision could be called bad.

While this approach may not be the only way to satisfy conditions A and B, it is the only one we know of. It assumes a method by which the generals communicate their values $v(i)$ to one another. The obvious method is for the ith general to send $v(i)$ by messenger to each other general. However, this does not work, because satisfying condition A requires that every loyal general obtain the same values $v(1), \ldots, v(n)$, and a traitorous general may send different values to different generals. For condition A to be satisfied, the following must be true:

1. Every loyal general must obtain the same information $v(1), \ldots, v(n)$.

Condition 1 implies that a general cannot necessarily use a value of $v(i)$ obtained directly from the ith general, since a traitorous ith general may send different values to different generals. This means that unless we are careful, in meeting condition 1 we might introduce the possibility that the generals use a value of $v(i)$ different from the one sent by the ith general—even though the ith general is loyal. We must not allow this to happen if condition B is to be met. For example, we cannot permit a few traitors to cause the loyal generals to base their decision upon the values "retreat", . . . , "retreat" if every loyal general sent the value "attack". We therefore have the following requirement for each i:

2. If the ith general is loyal, then the value that he sends must be used by every loyal general as the value of $v(i)$.

We can rewrite condition 1 as the condition that for every i (whether or not the ith general is loyal),

1′. Any two loyal generals use the same value of $v(i)$.

Conditions 1′ and 2 are both conditions on the single value sent by the ith general. We can therefore restrict our consideration to the problem of how a single general sends his value to the others. We phrase this in terms of a commanding general sending an order to his lieutenants, obtaining the following problem.

Byzantine Generals Problem. A commanding general must send an order to his $n - 1$ lieutenant generals such that

IC1. All loyal lieutenants obey the same order.
IC2. If the commanding general is loyal, then every loyal lieutenant obeys the order he sends.

Conditions IC1 and IC2 are called the *interactive consistency* conditions. Note that if the commander is loyal, then IC1 follows from IC2. However, the commander need not be loyal.

To solve our original problem, the ith general sends his value of $v(i)$ by using a solution to the Byzantine Generals Problem to send the order "use $v(i)$ as my value", with the other generals acting as the lieutenants.

2. IMPOSSIBILITY RESULTS

The Byzantine Generals Problem seems deceptively simple. Its difficulty is indicated by the surprising fact that if the generals can send only oral messages, then no solution will work unless more than two-thirds of the generals are loyal. In particular, with only three generals, no solution can work in the presence of a single traitor. An oral message is one whose contents are completely under the control of the sender, so a traitorous sender can transmit any possible message. Such a message corresponds to the type of message that computers normally send to one another. In Section 4 we consider signed, written messages, for which this is not true.

We now show that with oral messages no solution for three generals can handle a single traitor. For simplicity, we consider the case in which the only possible decisions are "attack" or "retreat". Let us first examine the scenario pictured in Figure 1 in which the commander is loyal and sends an "attack" order, but Lieutenant 2 is a traitor and reports to Lieutenant 1 that he received a "retreat" order. For IC2 to be satisfied, Lieutenant 1 must obey the order to attack.

Now consider another scenario, shown in Figure 2, in which the commander is a traitor and sends an "attack" order to Lieutenant 1 and a "retreat" order to Lieutenant 2. Lieutenant 1 does not know who the traitor is, and he cannot tell what message the commander actually sent to Lieutenant 2. Hence, the scenarios in these two pictures appear exactly the same to Lieutenant 1. If the traitor lies consistently, then there is no way for Lieutenant 1 to distinguish between these two situations, so he must obey the "attack" order in both of them. Hence, whenever Lieutenant 1 receives an "attack" order from the commander, he must obey it.

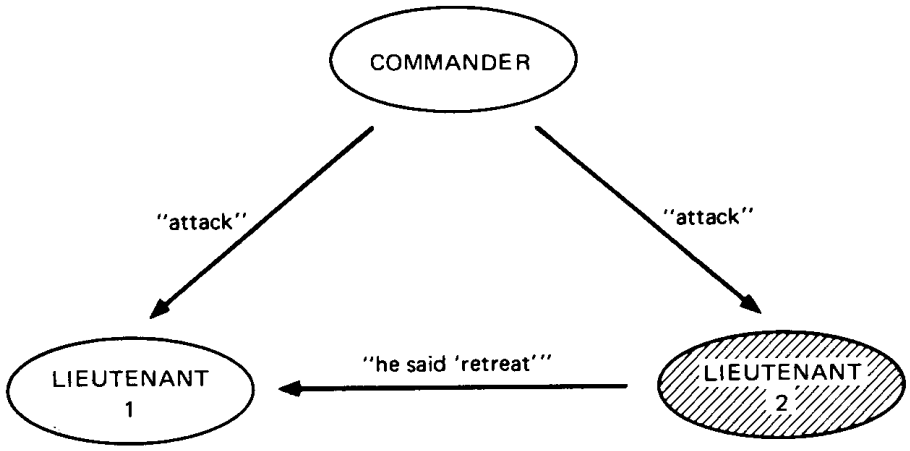


Fig. 1. Lieutenant 2 a traitor.

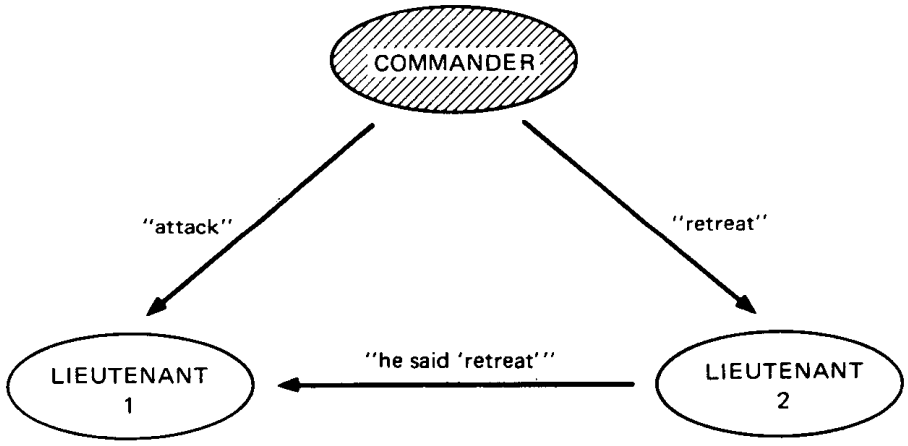


Fig. 2. The commander a traitor.

However, a similar argument shows that if Lieutenant 2 receives a "retreat" order from the commander then he must obey it even if Lieutenant 1 tells him that the commander said "attack". Therefore, in the scenario of Figure 2, Lieutenant 2 must obey the "retreat" order while Lieutenant 1 obeys the "attack" order, thereby violating condition IC1. Hence, no solution exists for three generals that works in the presence of a single traitor.

This argument may appear convincing, but we strongly advise the reader to be very suspicious of such nonrigorous reasoning. Although this result is indeed correct, we have seen equally plausible "proofs" of invalid results. We know of no area in computer science or mathematics in which informal reasoning is more likely to lead to errors than in the study of this type of algorithm. For a rigorous proof of the impossibility of a three-general solution that can handle a single traitor, we refer the reader to [3].

Using this result, we can show that no solution with fewer than $3m + 1$ generals can cope with m traitors.[1] The proof is by contradiction—we assume such a

[1] More precisely, no such solution exists for three or more generals, since the problem is trivial for two generals.

solution for a group of $3m$ or fewer and use it to construct a three-general solution to the Byzantine Generals Problem that works with one traitor, which we know to be impossible. To avoid confusion between the two algorithms, we call the generals of the assumed solution Albanian generals, and those of the constructed solution Byzantine generals. Thus, starting from an algorithm that allows $3m$ or fewer Albanian generals to cope with m traitors, we construct a solution that allows three Byzantine generals to handle a single traitor.

The three-general solution is obtained by having each of the Byzantine generals simulate approximately one-third of the Albanian generals, so that each Byzantine general is simulating at most m Albanian generals. The Byzantine commander simulates the Albanian commander plus at most $m - 1$ Albanian lieutenants, and each of the two Byzantine lieutenants simulates at most m Albanian lieutenants. Since only one Byzantine general can be a traitor, and he simulates at most m Albanians, at most m of the Albanian generals are traitors. Hence, the assumed solution guarantees that IC1 and IC2 hold for the Albanian generals. By IC1, all the Albanian lieutenants being simulated by a loyal Byzantine lieutenant obey the same order, which is the order he is to obey. It is easy to check that conditions IC1 and IC2 of the Albanian generals solution imply the corresponding conditions for the Byzantine generals, so we have constructed the required impossible solution.

One might think that the difficulty in solving the Byzantine Generals Problem stems from the requirement of reaching exact agreement. We now demonstrate that this is not the case by showing that reaching approximate agreement is just as hard as reaching exact agreement. Let us assume that instead of trying to agree on a precise battle plan, the generals must agree only upon an approximate time of attack. More precisely, we assume that the commander orders the time of the attack, and we require the following two conditions to hold:

IC1′. All loyal lieutenants attack within 10 minutes of one another.
IC2′. If the commanding general is loyal, then every loyal lieutenant attacks within 10 minutes of the time given in the commander's order.

(We assume that the orders are given and processed the day before the attack and that the time at which an order is received is irrelevant—only the attack time given in the order matters.)

Like the Byzantine Generals Problem, this problem is unsolvable unless more than two-thirds of the generals are loyal. We prove this by first showing that if there were a solution for three generals that coped with one traitor, then we could construct a three-general solution to the Byzantine Generals Problem that also worked in the presence of one traitor. Suppose the commander wishes to send an "attack" or "retreat" order. He orders an attack by sending an attack time of 1:00 and orders a retreat by sending an attack time of 2:00, using the assumed algorithm. Each lieutenant uses the following procedure to obtain his order.

(1) After receiving the attack time from the commander, a lieutenant does one of the following:

    (a) If the time is 1:10 or earlier, then attack.
    (b) If the time is 1:50 or later, then retreat.
    (c) Otherwise, continue to step (2).

(2) Ask the other lieutenant what decision he reached in step (1).

(a) If the other lieutenant reached a decision, then make the same decision he did.
(b) Otherwise, retreat.

It follows from IC2′ that if the commander is loyal, then a loyal lieutenant will obtain the correct order in step (1), so IC2 is satisfied. If the commander is loyal, then IC1 follows from IC2, so we need only prove IC1 under the assumption that the commander is a traitor. Since there is at most one traitor, this means that both lieutenants are loyal. It follows from IC1′ that if one lieutenant decides to attack in step (1), then the other cannot decide to retreat in step (1). Hence, either they will both come to the same decision in step (1) or at least one of them will defer his decision until step (2). In this case, it is easy to see that they both arrive at the same decision, so IC1 is satisfied. We have therefore constructed a three-general solution to the Byzantine Generals Problem that handles one traitor, which is impossible. Hence, we cannot have a three-general algorithm that maintains IC1′ and IC2′ in the presence of a traitor.

The method of having one general simulate m others can now be used to prove that no solution with fewer than $3m + 1$ generals can cope with m traitors. The proof is similar to the one for the original Byzantine Generals Problem and is left to the reader.

3. A SOLUTION WITH ORAL MESSAGES

We showed above that for a solution to the Byzantine Generals Problem using oral messages to cope with m traitors, there must be at least $3m + 1$ generals. We now give a solution that works for $3m + 1$ or more generals. However, we first specify exactly what we mean by "oral messages". Each general is supposed to execute some algorithm that involves sending messages to the other generals, and we assume that a loyal general correctly executes his algorithm. The definition of an oral message is embodied in the following assumptions which we make for the generals' message system:

A1. Every message that is sent is delivered correctly.
A2. The receiver of a message knows who sent it.
A3. The absence of a message can be detected.

Assumptions A1 and A2 prevent a traitor from interfering with the communication between two other generals, since by A1 he cannot interfere with the messages they do send, and by A2 he cannot confuse their intercourse by introducing spurious messages. Assumption A3 will foil a traitor who tries to prevent a decision by simply not sending messages. The practical implementation of these assumptions is discussed in Section 6.

The algorithms in this section and in the following one require that each general be able to send messages directly to every other general. In Section 5, we describe algorithms which do not have this requirement.

A traitorous commander may decide not to send any order. Since the lieutenants must obey some order, they need some default order to obey in this case. We let RETREAT be this default order.

We inductively define the *O*ral *M*essage algorithms OM(m), for all nonnegative integers m, by which a commander sends an order to $n - 1$ lieutenants. We show that OM(m) solves the Byzantine Generals Problem for $3m + 1$ or more generals in the presence of at most m traitors. We find it more convenient to describe this algorithm in terms of the lieutenants "obtaining a value" rather than "obeying an order".

The algorithm assumes a function *majority* with the property that if a majority of the values v_i equal v, then *majority*$(v_1, \ldots, v_{n-1})$ equals v. (Actually, it assumes a sequence of such functions—one for each n.) There are two natural choices for the value of *majority*$(v_1, \ldots, v_{n-1})$:

1. The majority value among the v_i if it exists, otherwise the value RETREAT;
2. The median of the v_i, assuming that they come from an ordered set.

The following algorithm requires only the aforementioned property of *majority*.

Algorithm OM(0).

(1) The commander sends his value to every lieutenant.
(2) Each lieutenant uses the value he receives from the commander, or uses the value RETREAT if he receives no value.

Algorithm OM(m), $m > 0$.

(1) The commander sends his value to every lieutenant.
(2) For each i, let v_i be the value Lieutenant i receives from the commander, or else be RETREAT if he receives no value. Lieutenant i acts as the commander in Algorithm OM($m - 1$) to send the value v_i to each of the $n - 2$ other lieutenants.
(3) For each i, and each $j \neq i$, let v_j be the value Lieutenant i received from Lieutenant j in step (2) (using Algorithm OM($m - 1$)), or else RETREAT if he received no such value. Lieutenant i uses the value *majority*$(v_1, \ldots, v_{n-1})$.

To understand how this algorithm works, we consider the case $m = 1$, $n = 4$. Figure 3 illustrates the messages received by Lieutenant 2 when the commander sends the value v and Lieutenant 3 is a traitor. In the first step of OM(1), the commander sends v to all three lieutenants. In the second step, Lieutenant 1 sends the value v to Lieutenant 2, using the trivial algorithm OM(0). Also in the second step, the traitorous Lieutenant 3 sends Lieutenant 2 some other value x. In step 3, Lieutenant 2 then has $v_1 = v_2 = v$ and $v_3 = x$, so he obtains the correct value $v =$ *majority*(v, v, x).

Next, we see what happens if the commander is a traitor. Figure 4 shows the values received by the lieutenants if a traitorous commander sends three arbitrary values x, y, and z to the three lieutenants. Each lieutenant obtains $v_1 = x$, $v_2 = y$, and $v_3 = z$, so they all obtain the same value *majority*(x, y, z) in step (3), regardless of whether or not any of the three values x, y, and z are equal.

The recursive algorithm OM(m) invokes $n - 1$ separate executions of the algorithm OM($m - 1$), each of which invokes $n - 2$ executions of OM($m - 2$), etc. This means that, for $m > 1$, a lieutenant sends many separate messages to each other lieutenant. There must be some way to distinguish among these different messages. The reader can verify that all ambiguity is removed if each lieutenant i prefixes the number i to the value v_i that he sends in step (2). As the recursion "unfolds," the algorithm OM($m - k$) will be called $(n - 1) \cdots (n - k)$ times to send a value prefixed by a sequence of k lieutenants' numbers.

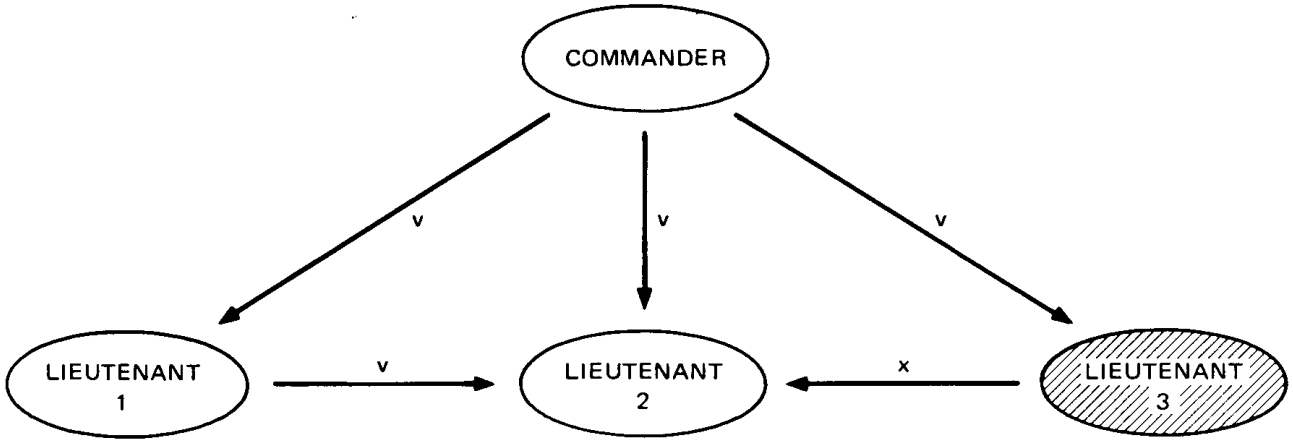


Fig. 3. Algorithm OM(1); Lieutenant 3 a traitor.

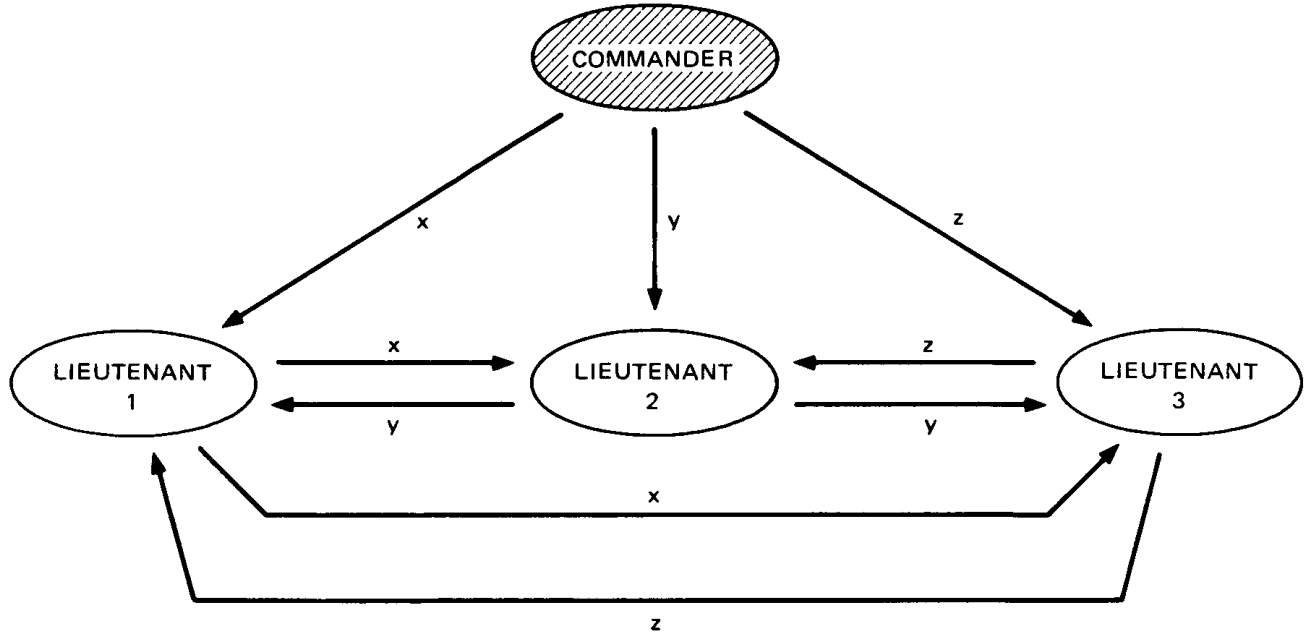


Fig. 4. Algorithm OM(1); the commander a traitor.

To prove the correctness of the algorithm OM(m) for arbitrary m, we first prove the following lemma.

LEMMA 1. *For any m and k, Algorithm OM(m) satisfies IC2 if there are more than $2k + m$ generals and at most k traitors.*

PROOF. The proof is by induction on m. IC2 only specifies what must happen if the commander is loyal. Using A1, it is easy to see that the trivial algorithm OM(0) works if the commander is loyal, so the lemma is true for $m = 0$. We now assume it is true for $m - 1$, $m > 0$, and prove it for m.

In step (1), the loyal commander sends a value v to all $n - 1$ lieutenants. In step (2), each loyal lieutenant applies OM($m - 1$) with $n - 1$ generals. Since by hypothesis $n > 2k + m$, we have $n - 1 > 2k + (m - 1)$, so we can apply the induction hypothesis to conclude that every loyal lieutenant gets $v_j = v$ for each loyal Lieutenant j. Since there are at most k traitors, and $n - 1 > 2k + (m - 1) \geq 2k$, a majority of the $n - 1$ lieutenants are loyal. Hence, each loyal lieutenant has $v_i = v$ for a majority of the $n - 1$ values i, so he obtains *majority*$(v_1, \ldots, v_{n-1}) = v$ in step (3), proving IC2. □

The following theorem asserts that Algorithm OM(m) solves the Byzantine Generals Problem.

THEOREM 1. *For any m, Algorithm OM(m) satisfies conditions IC1 and IC2 if there are more than $3m$ generals and at most m traitors.*

PROOF. The proof is by induction on m. If there are no traitors, then it is easy to see that OM(0) satisfies IC1 and IC2. We therefore assume that the theorem is true for OM($m - 1$) and prove it for OM(m), $m > 0$.

We first consider the case in which the commander is loyal. By taking k equal to m in Lemma 1, we see that OM(m) satisfies IC2. IC1 follows from IC2 if the commander is loyal, so we need only verify IC1 in the case that the commander is a traitor.

There are at most m traitors, and the commander is one of them, so at most $m - 1$ of the lieutenants are traitors. Since there are more than $3m$ generals, there are more than $3m - 1$ lieutenants, and $3m - 1 > 3(m - 1)$. We may therefore apply the induction hypothesis to conclude that OM($m - 1$) satisfies conditions IC1 and IC2. Hence, for each j, any two loyal lieutenants get the same value for v_j in step (3). (This follows from IC2 if one of the two lieutenants is Lieutenant j, and from IC1 otherwise.) Hence, any two loyal lieutenants get the same vector of values $v_1, \ldots, v_{n-1}$, and therefore obtain the same value *majority*$(v_1, \ldots, v_{n-1})$ in step (3), proving IC1. □

4. A SOLUTION WITH SIGNED MESSAGES

As we saw from the scenario of Figures 1 and 2, it is the traitors' ability to lie that makes the Byzantine Generals Problem so difficult. The problem becomes easier to solve if we can restrict that ability. One way to do this is to allow the generals to send unforgeable signed messages. More precisely, we add to A1–A3 the

following assumption:

A4 (a) A loyal general's signature cannot be forged, and any alteration of the contents of his signed messages can be detected.
(b) Anyone can verify the authenticity of a general's signature.

Note that we make no assumptions about a traitorous general's signature. In particular, we allow his signature to be forged by another traitor, thereby permitting collusion among the traitors.

Now that we have introduced signed messages, our previous argument that four generals are required to cope with one traitor no longer holds. In fact, a three-general solution does exist. We now give an algorithm that copes with m traitors for any number of generals. (The problem is vacuous if there are fewer than $m + 2$ generals.)

In our algorithm, the commander sends a signed order to each of his lieutenants. Each lieutenant then adds his signature to that order and sends it to the other lieutenants, who add their signatures and send it to others, and so on. This means that a lieutenant must effectively receive one signed message, make several copies of it, and sign and send those copies. It does not matter how these copies are obtained; a single message might be photocopied, or else each message might consist of a stack of identical messages which are signed and distributed as required.

Our algorithm assumes a function *choice* which is applied to a set of orders to obtain a single one. The only requirements we make for this function are

1. If the set V consists of the single element v, then $choice(V) = v$.
2. $choice(\varnothing) =$ RETREAT, where $\varnothing$ is the empty set.

Note that one possible definition is to let $choice(V)$ be the median element of V—assuming that there is an ordering of the elements.

In the following algorithm, we let $x:i$ denote the value x signed by General i. Thus, $v:j:i$ denotes the value v signed by j, and then that value $v:j$ signed by i. We let General 0 be the commander. In this algorithm, each lieutenant i maintains a set V_i, containng the set of properly signed orders he has received so far. (If the commander is loyal, then this set should never contain more than a single element.) Do not confuse V_i, the set of *orders* he has received, with the set of messages that he has received. There may be many different messages with the same order.

Algorithm SM(m).

Initially $V_i = \varnothing$.

(1) The commander signs and sends his value to every lieutenant.
(2) For each i:

(A) If Lieutenant i receives a message of the form $v:0$ from the commander and he has not yet received any order, then

(i) he lets V_i equal $\{v\}$;
(ii) he sends the message $v:0:i$ to every other lieutenant.

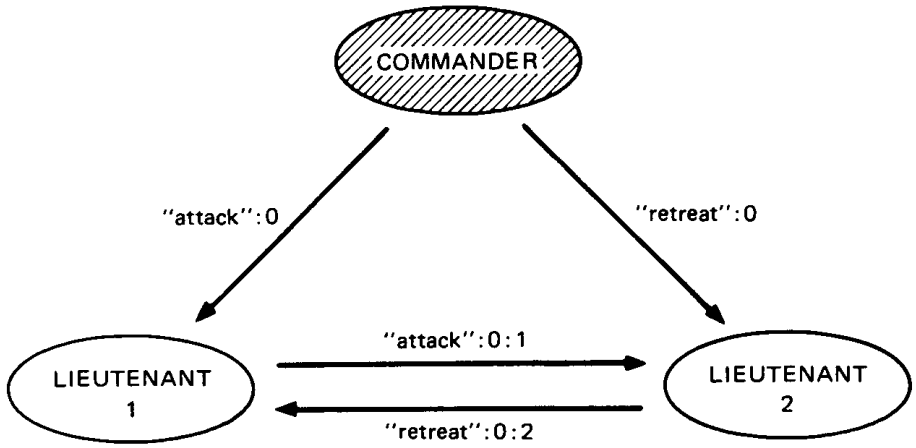


Fig. 5. Algorithm SM(1); the commander a traitor.

(B) If Lieutenant i receives a message of the form $v:0:j_1:\cdots:j_k$ and v is not in the set V_i, then

(i) he adds v to V_i;

(ii) if $k < m$, then he sends the message $v:0:j_1:\cdots:j_k:i$ to every lieutenant other than $j_1, \ldots, j_k$.

(3) For each i: When Lieutenant i will receive no more messages, he obeys the order $choice(V_i)$.

Note that in step (2), Lieutenant i ignores any message containing an order v that is already in the set V_i.

We have not specified how a lieutenant determines in step (3) that he will receive no more messages. By induction on k, one easily shows that for each sequence of lieutenants $j_1, \ldots, j_k$ with $k \leq m$, a lieutenant can receive at most one message of the form $v:0:j_1:\cdots:j_k$ in step (2). If we require that Lieutenant j_k either send such a message or else send a message reporting that he will not send such a message, then it is easy to decide when all messages have been received. (By assumption A3, a lieutenant can determine if a traitorous lieutenant j_k sends neither of those two messages.) Alternatively, time-out can be used to determine when no more messages will arrive. The use of time-out is discussed in Section 6.

Note that in step (2), Lieutenant i ignores any messages that do not have the proper form of a value followed by a string of signatures. If packets of identical messages are used to avoid having to copy messages, this means that he throws away any packet that does not consist of a sufficient number of identical, properly signed messages. (There should be $(n-k-2)(n-k-3)\cdots(n-m-2)$ copies of the message if it has been signed by k lieutenants.)

Figure 5 illustrates Algorithm SM(1) for the case of three generals when the commander is a traitor. The commander sends an "attack" order to one lieutenant and a "retreat" order to the other. Both lieutenants receive the two orders in step (2), so after step (2) $V_1 = V_2 = \{$"attack", "retreat"$\}$, and they both obey the order $choice(\{$"attack", "retreat"$\})$. Observe that here, unlike the situation in Figure 2, the lieutenants know the commander is a traitor because his signature

appears on two different orders, and A4 states that only he could have generated those signatures.

In Algorithm SM(m), a lieutenant signs his name to acknowledge his receipt of an order. If he is the mth lieutenant to add his signature to the order, then that signature is not relayed to anyone else by its recipient, so it is superfluous. (More precisely, assumption A2 makes it unnecessary.) In particular, the lieutenants need not sign their messages in SM(1).

We now prove the correctness of our algorithm.

THEOREM 2. *For any m, Algorithm SM(m) solves the Byzantine Generals Problem if there are at most m traitors.*

PROOF. We first prove IC2. If the commander is loyal, then he sends his signed order $v:0$ to every lieutenant in step (1). Every loyal lieutenant will therefore receive the order v in step (2)(A). Moreover, since no traitorous lieutenant can forge any other message of the form $v':0$, a loyal lieutenant can receive no additional order in step (2)(B). Hence, for each loyal Lieutenant i, the set V_i obtained in step (2) consists of the single order v, which he will obey in step (3) by property 1 of the *choice* function. This proves IC2.

Since IC1 follows from IC2 if the commander is loyal, to prove IC1 we need only consider the case in which the commander is a traitor. Two loyal lieutenants i and j obey the same order in step (3) if the sets of orders V_i and V_j that they receive in step (2) are the same. Therefore, to prove IC1 it suffices to prove that, if i puts an order v into V_i in step (2), then j must put the same order v into V_j in step (2). To do this, we must show that j receives a properly signed message containing that order. If i receives the order v in step (2)(A), then he sends it to j in step (2)(A)(ii); so j receives it (by A1). If i adds the order to V_i in step (2)(B), then he must receive a first message of the form $v:0:j_1:\cdots:j_k$. If j is one of the j_r, then by A4 he must already have received the order v. If not, we consider two cases:

1. $k < m$. In this case, i sends the message $v:0:j_1:\cdots:j_k:i$ to j; so j must receive the order v.
2. $k = m$. Since the commander is a traitor, at most $m - 1$ of the lieutenants are traitors. Hence, at least one of the lieutenants $j_1, \ldots, j_m$ is loyal. This loyal lieutenant must have sent j the value v when he first received it, so j must therefore receive that value.

This completes the proof. □

5. MISSING COMMUNICATION PATHS

Thus far, we have assumed that a general can send messages directly to every other general. We now remove this assumption. Instead, we suppose that physical barriers place some restrictions on who can send messages to whom. We consider the generals to form the nodes of a simple,[2] finite undirected graph G, where an arc between two nodes indicates that those two generals can send messages

[2] A simple graph is one in which there is at most one arc joining any two nodes, and every arc connects two distinct nodes.

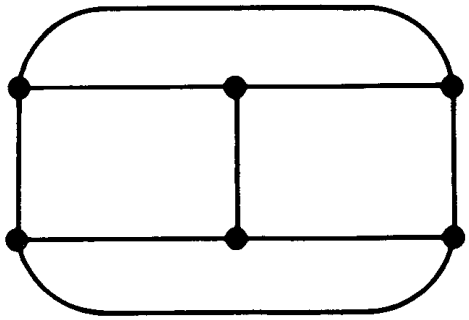


Fig. 6. A 3-regular graph.

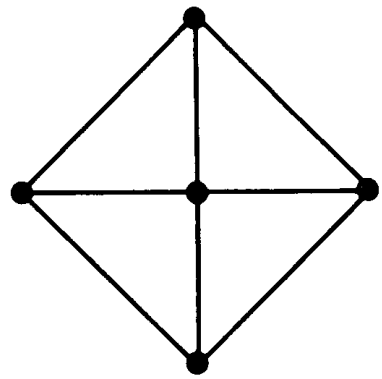


Fig. 7. A graph that is not 3-regular.

directly to one another. We now extend Algorithms OM(m) and SM(m), which assumed G to be completely connected, to more general graphs.

To extend our oral message algorithm OM(m), we need the following definition, where two generals are said to be *neighbors* if they are joined by an arc.

Definition 1.

(a) A set of nodes $\{i_1, \ldots, i_p\}$ is said to be a *regular set of neighbors* of a node i if

(i) each i_j is a neighbor of i, and
(ii) for any general k different from i, there exist paths $\gamma_{j,k}$ from i_j to k not passing through i such that any two different paths $\gamma_{j,k}$ have no node in common other than k.

(b) The graph G is said to be *p-regular* if every node has a regular set of neighbors consisting of p distinct nodes.

Figure 6 shows an example of a simple 3-regular graph. Figure 7 shows an example of a graph that is not 3-regular because the central node has no regular set of neighbors containing three nodes.

We extend OM(m) to an algorithm that solves the Byzantine Generals Problem in the presence of m traitors if the graph G of generals is $3m$-regular. (Note that a $3m$-regular graph must contain at least $3m + 1$ nodes.) For all *positive* integers m and p, we define the algorithm OM(m, p) as follows when the graph G of generals is p-regular. (OM(m, p) is not defined if G is not p-regular.) The definition uses induction on m.

Algorithm *OM*(m, p).

(0) Choose a regular set N of neighbors of the commander consisting of p lieutenants.
(1) The commander sends his value to every lieutenant in N.
(2) For each i in N, let v_i be the value Lieutenant i receives from the commander, or else RETREAT if he receives no value. Lieutenant i sends v_i to every other lieutenant k as follows:

  (A) If $m = 1$, then by sending the value along the path $\gamma_{i,k}$ whose existence is guaranteed by part (a)(ii) of Definition 1.
  (B) If $m > 1$, then by acting as the commander in the algorithm OM($m - 1, p - 1$), with the graph of generals obtained by removing the original commander from G.

(3) For each k, and each i in N with $i \neq k$, let v_i be the value Lieutenant k received from Lieutenant i in step (2), or RETREAT if he received no value. Lieutenant k uses the value $majority(v_{i_1}, \ldots, v_{i_p})$, where $N = \{i_1, \ldots, i_p\}$.

Note that removing a single node from a p-regular graph leaves a $(p - 1)$-regular graph. Hence, one can apply the algorithm OM$(m - 1, p - 1)$ in step (2)(B).

We now prove that OM$(m, 3m)$ solves the Byzantine Generals Problem if there are at most m traitors. The proof is similar to the proof for the algorithm OM(m) and will just be sketched. It begins with the following extension of Lemma 1.

LEMMA 2. *For any $m > 0$ and any $p \geq 2k + m$, Algorithm OM(m, p) satisfies IC2 if there are at most k traitors.*

PROOF. For $m = 1$, observe that a lieutenant obtains the value $majority(v_1, \ldots, v_p)$, where each v_i is a value sent to him by the commander along a path disjoint from the path used to send the other values to him. Since there are at most k traitors and $p \geq 2k + 1$, more than half of those paths are composed entirely of loyal lieutenants. Hence, if the commander is loyal, then a majority of the values v_i will equal the value he sent, which implies that IC2 is satisfied.

Now assume the lemma for $m - 1$, $m > 1$. If the commander is loyal, then each of the p lieutenants in N gets the correct value. Since $p > 2k$, a majority of them are loyal, and by the induction hypothesis each of them sends the correct value to every loyal lieutenant. Hence, each loyal lieutenant gets a majority of correct values, thereby obtaining the correct value in step (3). □

The correctness of Algorithm OM$(m, 3m)$ is an immediate consequence of the following result.

THEOREM 3. *For any $m > 0$ and any $p \geq 3m$, Algorithm OM(m, p) solves the Byzantine Generals Problem if there are at most m traitors.*

PROOF. By Lemma 2, letting $k = m$, we see that OM(m, p) satisfies IC2. If the commander is loyal, then IC1 follows from IC2, so we need only prove IC1 under the assumption that the commander is a traitor. To do this, we prove that every loyal lieutenant gets the same set of values v_i in step (3). If $m = 1$, then this follows because all the lieutenants, including those in N, are loyal and the paths $\gamma_{i,k}$ do not pass through the commander. For $m > 1$, a simple induction argument can be applied, since $p \geq 3m$ implies that $p - 1 \geq 3(m - 1)$. □

Our extension of Algorithm OM(m) requires that the graph G be $3m$-regular, which is a rather strong connectivity hypothesis.[3] In fact, if there are only $3m + 1$ generals (the minimum number required), then $3m$-regularity means complete connectivity, and Algorithm OM$(m, 3m)$ reduces to Algorithm OM(m). In contrast, Algorithm SM(m) is easily extended to allow the weakest possible connectivity hypothesis. Let us first consider how much connectivity is needed for the Byzantine Generals Problem to be solvable. IC2 requires that a loyal lieutenant obey a loyal commander. This is clearly impossible if the commander cannot communicate with the lieutenant. In particular, if every message from the

[3] A recent algorithm of Dolev [2] requires less connectivity.

commander to the lieutenant must be relayed by traitors, then there is no way to guarantee that the lieutenant gets the commander's order. Similarly, IC1 cannot be guaranteed if there are two lieutenants who can only communicate with one another via traitorous intermediaries.

The weakest connectivity hypothesis for which the Byzantine Generals Problem is solvable is that the subgraph formed by the loyal generals be connected. We show that under this hypothesis, the algorithm SM($n - 2$) is a solution, where n is the number of generals—regardless of the number of traitors. Of course, we must modify the algorithm so that generals only send messages to where they can be sent. More precisely, in step (1), the commander sends his signed order only to his neighboring lieutenants; and in step (2)(B), Lieutenant i only sends the message to every *neighboring* lieutenant not among the j_r.

We prove the following more general result, where the *diameter* of a graph is the smallest number d such that any two nodes are connected by a path containing at most d arcs.

THEOREM 4. *For any m and d, if there are at most m traitors and the subgraph of loyal generals has diameter d, then Algorithm SM($m + d - 1$) (with the above modification) solves the Byzantine Generals Problem.*

PROOF. The proof is quite similar to that of Theorem 2 and is just sketched here. To prove IC2, observe that by hypothesis there is a path from the loyal commander to a lieutenant i going through $d - 1$ or fewer loyal lieutenants. Those lieutenants will correctly relay the order until it reaches i. As before, assumption A4 prevents a traitor from forging a different order.

To prove IC1, we assume the commander is a traitor and must show that any order received by a loyal lieutenant i is also received by a loyal lieutenant j. Suppose i receives an order $v:0:j_1: \cdots :j_k$ not signed by j. If $k < m$, then i will send it to every neighbor who has not already received that order, and it will be relayed to j within $d - 1$ more steps. If $k \geq m$, then one of the first m signers must be loyal and must have sent it to all of his neighbors, whereupon it will be relayed by loyal generals and will reach j within $d - 1$ steps. □

COROLLARY. *If the graph of loyal generals is connected, then SM($n - 2$) (as modified above) solves the Byzantine Generals Problem for n generals.*

PROOF. Let d be the diameter of the graph of loyal generals. Since the diameter of a connected graph is less than the number of nodes, there must be more than d loyal generals and fewer than $n - d$ traitors. The result follows from the theorem by letting $m = n - d - 1$. □

Theorem 4 assumes that the subgraph of loyal generals is connected. Its proof is easily extended to show that even if this is not the case, if there are at most m traitors, then the algorithm SM($m + d - 1$) has the following two properties:

1. Any two loyal generals connected by a path of length at most d passing through only loyal generals will obey the same order.
2. If the commander is loyal, then any loyal lieutenant connected to him by a path of length at most $m + d$ passing only through loyal generals will obey his order.

6. RELIABLE SYSTEMS

Other than using intrinsically reliable circuit components, the only way we know to implement a reliable computer system is to use several different "processors" to compute the same result, and then to perform a majority vote on their outputs to obtain a single value. (The voting may be performed within the system, or externally by the users of the output.) This is true whether one is implementing a reliable computer using redundant circuitry to protect against the failure of individual chips, or a ballistic missile defense system using redundant computing sites to protect against the destruction of individual sites by a nuclear attack. The only difference is in the size of the replicated "processor".

The use of majority voting to achieve reliability is based upon the assumption that all the nonfaulty processors will produce the same output. This is true so long as they all use the same input. However, any single input datum comes from a single physical component—for example, from some other circuit in the reliable computer, or from some radar site in the missile defense system—and a malfunctioning component can give different values to different processors. Moreover, different processors can get different values even from a nonfaulty input unit if they read the value while it is changing. For example, if two processors read a clock while it is advancing, then one may get the old time and the other the new time. This can only be prevented by synchronizing the reads with the advancing of the clock.

In order for majority voting to yield a reliable system, the following two conditions should be satisfied:

1. All nonfaulty processors must use the same input value (so they produce the same output).
2. If the input unit is nonfaulty, then all nonfaulty processes use the value it provides as input (so they produce the correct output).

These are just our interactive consistency conditions IC1 and IC2, where the "commander" is the unit generating the input, the "lieutenants" are the processors, and "loyal" means nonfaulty.

It is tempting to try to circumvent the problem with a "hardware" solution. For example, one might try to insure that all processors obtain the same input value by having them all read it from the same wire. However, a faulty input unit could send a marginal signal along the wire—a signal that can be interpreted by some processors as a 0 and by others as a 1. There is no way to guarantee that different processors will get the same value from a possibly faulty input device except by having the processors communicate among themselves to solve the Byzantine Generals Problem.

Of course, a faulty input device may provide meaningless input values. All that a Byzantine Generals solution can do is guarantee that all processors use the same input value. If the input is an important one, then there should be several separate input devices providing redundant values. For example, there should be redundant radars as well as redundant processing sites in a missile defense system. However, redundant inputs cannot achieve reliability; it is still necessary to insure that the nonfaulty processors use the redundant data to produce the same output.

In case the input device is nonfaulty but gives different values because it is read while its value is changing, we still want the nonfaulty processors to obtain a reasonable input value. It can be shown that, if the functions *majority* and *choice* are taken to be the median functions, then our algorithms have the property that the value obtained by the nonfaulty processors lies within the range of values provided by the input unit. Thus, the nonfaulty processors will obtain a reasonable value so long as the input unit produces a reasonable range of values.

We have given several solutions, but they have been stated in terms of Byzantine generals rather than in terms of computing systems. We now examine how these solutions can be applied to reliable computing systems. Of course, there is no problem implementing a "general's" algorithm with a processor. The problems lie in implementing a message passing system that meets assumptions A1–A3 (assumptions A1–A4 for Algorithm SM(m)). We now consider these assumptions in order.

A1. Assumption A1 states that every message sent by a nonfaulty processor is delivered correctly. In real systems, communication lines can fail. For the oral message algorithms OM(m) and OM(m, p), the failure of the communication line joining two processors is indistinguishable from the failure of one of the processors. Hence, we can only guarantee that these algorithms will work in the presence of up to m failures, be they processor or communication line failures. (Of course, the failure of several communication lines attached to the same processor is equivalent to a single processor failure.) If we assume that a failed communication line cannot result in the forgery of a signed message—an assumption which we will see below is quite reasonable, then our signed message algorithm SM(m) is insensitive to communication line failure. More precisely, Theorem 4 remains valid even with communication line failure. A failed communication line has the same effect as simply removing the communication line—it lowers the connectivity of the processor graph.

A2. Assumption A2 states that a processor can determine the originator of any message that it received. What is actually necessary is that a faulty processor not be able to impersonate a nonfaulty one. In practice, this means that interprocess communication be over fixed lines rather than through some message switching network. (If a switching network is used, then faulty network nodes must be considered, and the Byzantine Generals Problem appears again.) Note that assumption A2 is not needed if A4 is assumed and all messages are signed, since impersonation of another processor would imply forging its messages.

A3. Assumption A3 requires that the absence of a message can be detected. The absence of a message can only be detected by its failure to arrive within some fixed length of time—in other words, by the use of some time-out convention. The use of time-out to satisfy A3 requires two assumptions:

1. There is a fixed maximum time needed for the generation and transmission of a message.
2. The sender and receiver have clocks that are synchronized to within some fixed maximum error.

The need for the first assumption is fairly obvious, since the receiver must know

how long he needs to wait for the message to arrive. (The generation time is how long it takes the processor to send the message after receiving all the input necessary to generate it.) The need for the second assumption is less obvious. However, it can be shown that either this assumption or an equivalent one is necessary to solve the Byzantine Generals Problem. More precisely, suppose that we allow algorithms in which the generals take action only in the following circumstances:

1. At some fixed initial time (the same for all generals).
2. Upon the receipt of a message.
3. When a randomly chosen length of time has elapsed. (I.e., a general can set a timer to a random value and act when the timer goes off.)

(This yields the most general class of algorithms we can envision which does not allow the construction of synchronized clocks.) It can be shown that no such algorithm can solve the Byzantine Generals Problem if messages can be transmitted arbitrarily quickly, even if there is an upper bound on message transmission delay. Moreover, no solution is possible even if we restrict the traitors so that the only incorrect behavior they are permitted is the failure to send a message. The proof of this result is beyond the scope of this paper. Note that placing a lower as well as an upper bound on transmission delay allows processors to implement clocks by sending messages back and forth.

The above two assumptions make it easy to detect unsent messages. Let μ be the maximum message generation and transmission delay, and assume the nonfaulty processors have clocks that differ from one another by at most τ at any time. Then any message that a nonfaulty process should begin to generate by time T on its clock will arrive at its destination by time $T + \mu + \tau$ on the receiver's clock. Hence, if the receiver has not received the message by that time, then it may assume that it was not sent. (If it arrives later, then the sender must be faulty, so the correctness of our algorithms does not depend upon the message being sent.) By fixing the time at which the input processor sends its value, one can calculate until what time on its own clock a processor must wait for each message. For example, in Algorithm SM(m) a processor must wait until time $T_0 + k(\mu + \tau)$ for any message having k signatures, where T_0 is the time (on his clock) at which the commander starts executing the algorithm.

No two clocks run at precisely the same rate, so no matter how accurately the processors' clocks are synchronized initially, they will eventually drift arbitrarily far apart unless they are periodically resynchronized. We therefore have the problem of keeping the processors' clocks all synchronized to within some fixed amount, even if some of the processors are faulty. This is as difficult a problem as the Byzantine Generals Problem itself. Solutions to the clock synchronization problem exist which are closely related to our Byzantine Generals solutions. They will be described in a future paper.

A4. Assumption A4 requires that processors be able to sign their messages in such a way that a nonfaulty processor's signature cannot be forged. A signature is a piece of redundant information $S_i(M)$ generated by process i from a data item M. A message signed by i consists of a pair $(M, S_i(M))$. To meet parts (a)

and (b) of A4, the function S_i must have the following two properties:

(a) If processor i is nonfaulty, then no faulty processor can generate $S_i(M)$.
(b) Given M and X, any process can determine if X equals $S_i(M)$.

Property (a) can never be guaranteed, since $S_i(M)$ is just a data item, and a faulty processor could generate any data item. However, we can make the probability of its violation as small as we wish, thereby making the system as reliable as we wish. How this is done depends upon the type of faults we expect to encounter. There are two cases of interest:

1. *Random Malfunction.* By making S_i a suitably "randomizing" function, we can make the probability that a random malfunction in a processor generates a correct signature essentially equal to the probability of its doing so through a random choice procedure—that is, the reciprocal of the number of possible signatures. The following is one method for doing this. Assume that messages are encoded as positive integers less than P, where P is a power of two. Let $S_i(M)$ equal $M * K_i \bmod P$, where K_i is a randomly chosen odd number less than P. Letting K_i^{-1} be the unique number less than P such that $K_i * K_i^{-1} \equiv 1 \bmod P$, a process can check that $X = S_i(M)$ by testing that $M \equiv X * K_i^{-1} \bmod P$. If another processor does not have K_i in its memory, then the probability of its generating the correct signature $M * K_i$ for a single (nonzero) message M should be $1/P$: its probability of doing so by random choice. (Note that if the processor could obtain K_i by some simple procedure, then there might be a larger probability of a faulty processor j forging i's signature by substituting K_i for K_j when trying to compute $S_j(M)$.)

2. *Malicious Intelligence.* If the faulty processor is being guided by a malicious intelligence—for example, if it is a perfectly good processor being operated by a human who is trying to disrupt the system—then the construction of the signature function S_i becomes a cryptography problem. We refer the reader to [1] and [4] for a discussion of how this problem can be solved.

Note that it is easy to generate the signature $S_i(M)$ if the process has already seen that signature. Hence, it is important that the same message never have to be signed twice. This means that, when using SM(m) repeatedly to distribute a sequence of values, sequence numbers should be appended to the values to guarantee uniqueness.

7. CONCLUSION

We have presented several solutions to the Byzantine Generals Problem, under various hypotheses, and shown how they can be used in implementing reliable computer systems. These solutions are expensive in both the amount of time and the number of messages required. Algorithms OM(m) and SM(m) both require message paths of length up to $m + 1$. In other words, each lieutenant may have to wait for messages that originated at the commander and were then relayed via m other lieutenants. Fischer and Lynch have shown that this must be true for any solution that can cope with m traitors, so our solutions are optimal in that respect. Our algorithms for a graph that is not completely connected require

message paths of length up to $m + d$, where d is the diameter of the subgraph of loyal generals. We suspect that this is also optimal.

Algorithms OM(m) and SM(m) involve sending up to $(n - 1)(n - 2) \cdots (n - m - 1)$ messages. The number of separate messages required can certainly be reduced by combining messages. It may also be possible to reduce the amount of information transferred, but this has not been studied in detail. However, we expect that a large number of messages will still be required.

Achieving reliability in the face of arbitrary malfunctioning is a difficult problem, and its solution seems to be inherently expensive. The only way to reduce the cost is to make assumptions about the type of failure that may occur. For example, it is often assumed that a computer may fail to respond but will never respond incorrectly. However, when extremely high reliability is required, such assumptions cannot be made, and the full expense of a Byzantine Generals solution is required.

REFERENCES

1. Diffie, W., and Hellman, M.E. New directions in cryptography. *IEEE Trans. Inf. Theory IT-22* (Nov. 1976), 644–654.
2. Dolev, D. The Byzantine generals strike again. *J. Algorithms 3*, 1 (Jan. 1982).
3. Pease, M., Shostak, R., and Lamport, L. Reaching agreement in the presence of faults. *J. ACM 27*, 2 (Apr. 1980), 228–234.
4. Rivest, R.L., Shamir, A., and Adleman, L. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM 21*, 2 (Feb. 1978), 120–126.

Received April 1980; revised November 1981; accepted November 1981

Chapter 5

The Part-Time Parliament

Authors:
Leslie Lamport

Publication year: 1998

The Part-Time Parliament

Leslie Lamport

This article appeared in *ACM Transactions on Computer Systems 16*, 2 (May 1998), 133-169. Minor corrections were made on 29 August 2000.

The Part-Time Parliament

LESLIE LAMPORT
Digital Equipment Corporation

Recent archaeological discoveries on the island of Paxos reveal that the parliament functioned despite the peripatetic propensity of its part-time legislators. The legislators maintained consistent copies of the parliamentary record, despite their frequent forays from the chamber and the forgetfulness of their messengers. The Paxon parliament's protocol provides a new way of implementing the state-machine approach to the design of distributed systems.

Categories and Subject Descriptors: C2.4 [**Computer-Communications Networks**]: Distributed Systems—*Network operating systems*; D4.5 [**Operating Systems**]: Reliability—*Fault-tolerance*; J.1 [**Administrative Data Processing**]: Government

General Terms: Design, Reliability

Additional Key Words and Phrases: State machines, three-phase commit, voting

This submission was recently discovered behind a filing cabinet in the *TOCS* editorial office. Despite its age, the editor-in-chief felt that it was worth publishing. Because the author is currently doing field work in the Greek isles and cannot be reached, I was asked to prepare it for publication.

The author appears to be an archeologist with only a passing interest in computer science. This is unfortunate; even though the obscure ancient Paxon civilization he describes is of little interest to most computer scientists, its legislative system is an excellent model for how to implement a distributed computer system in an asynchronous environment. Indeed, some of the refinements the Paxons made to their protocol appear to be unknown in the systems literature.

The author does give a brief discussion of the Paxon Parliament's relevance to distributed computing in Section 4. Computer scientists will probably want to read that section first. Even before that, they might want to read the explanation of the algorithm for computer scientists by Lampson [1996]. The algorithm is also described more formally by De Prisco et al. [1997]. I have added further comments on the relation between the ancient protocols and more recent work at the end of Section 4.

Keith Marzullo
University of California, San Diego

Authors' address: Systems Research Center, Digital Equipment Corporation, 130 Lytton Avenue, Palo Alto, CA 94301.
Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
© 1998 ACM 0000-0000/98/0000-0000 $00.00

1 The Problem

1.1 The Island of Paxos

Early in this millennium, the Aegean island of Paxos was a thriving mercantile center.[1] Wealth led to political sophistication, and the Paxons replaced their ancient theocracy with a parliamentary form of government. But trade came before civic duty, and no one in Paxos was willing to devote his life to Parliament. The Paxon Parliament had to function even though legislators continually wandered in and out of the parliamentary Chamber.

The problem of governing with a part-time parliament bears a remarkable correspondence to the problem faced by today's fault-tolerant distributed systems, where legislators correspond to processes and leaving the Chamber corresponds to failing. The Paxons' solution may therefore be of some interest to computer scientists. I present here a short history of the Paxos Parliament's protocol, followed by an even shorter discussion of its relevance for distributed systems.

Paxon civilization was destroyed by a foreign invasion, and archeologists have just recently begun to unearth its history. Our knowledge of the Paxon Parliament is therefore fragmentary. Although the basic protocols are known, we are ignorant of many details. Where such details are of interest, I will take the liberty of speculating on what the Paxons might have done.

1.2 Requirements

Parliament's primary task was to determine the law of the land, which was defined by the sequence of decrees it passed. A modern parliament will employ a secretary to record its actions, but no one in Paxos was willing to remain in the Chamber throughout the session to act as secretary. Instead, each Paxon legislator maintained a *ledger* in which he recorded the numbered sequence of decrees that were passed. For example, legislator $\Lambda\check{\imath}\nu\chi\partial$'s ledger had the entry

155: *The olive tax is 3 drachmas per ton*

if she believed that the 155th decree passed by Parliament set the tax on olives to 3 drachmas per ton. Ledgers were written with indelible ink, and their entries could not be changed.

The first requirement of the parliamentary protocol was the *consistency of ledgers*, meaning that no two ledgers could contain contradictory information. If legislator $\Phi\iota\sigma\partial\epsilon\rho$ had the entry

132: *Lamps must use only olive oil*

in his ledger, then no other legislator's ledger could have a different entry for decree 132. However, another legislator might have no entry in his ledger for decree 132 if he hadn't yet learned that the decree had been passed.

Consistency of ledgers was not sufficient, since it could be trivially fulfilled by leaving all ledgers blank. Some requirement was needed to guarantee that decrees

[1] It should not be confused with the Ionian island of Paxoi, whose name is sometimes corrupted to *Paxos*.

were eventually passed and recorded in ledgers. In modern parliaments, the passing of decrees is hindered by disagreement among legislators. This was not the case in Paxos, where an atmosphere of mutual trust prevailed. Paxon legislators were willing to pass any decree that was proposed. However, their peripatetic propensity posed a problem. Consistency would be lost if one group of legislators passed the decree

37: *Painting on temple walls is forbidden*

and then left for a banquet, whereupon a different group of legislators entered the Chamber and, knowing nothing about what had just happened, passed the conflicting decree

37: *Freedom of artistic expression is guaranteed*

Progress could not be guaranteed unless enough legislators stayed in the Chamber for a long enough time. Because Paxon legislators were unwilling to curtail their outside activities, it was impossible to ensure that any decree would ever be passed. However, legislators were willing to guarantee that, while in the Chamber, they and their aides would act promptly on all parliamentary matters. This guarantee allowed the Paxons to devise a parliamentary protocol satisfying the following *progress condition*.

> If a majority of the legislators[2] were in the Chamber and no one entered or left the Chamber for a sufficiently long period of time then any decree proposed by a legislator in the Chamber would be passed, and every decree that had been passed would appear in the ledger of every legislator in the Chamber.

1.3 Assumptions

The requirements of the parliamentary protocol could be achieved only by providing the legislators with the necessary resources. Each legislator received a sturdy ledger in which to record the decrees, a pen, and a supply of indelible ink. Legislators might forget what they had been doing if they left the Chamber,[3] so they would write notes in the back of the ledgers to remind themselves of important parliamentary tasks. An entry in the list of decrees was never changed, but notes could be crossed out. Achieving the progress condition required that legislators be able to measure the passage of time, so they were given simple hourglass timers.

Legislators carried their ledgers at all times, and could always read the list of decrees and any note that had not been crossed out. The ledgers were made of the finest parchment and were used for only the most important notes. A legislator would write other notes on a slip of paper, which he might (or might not) lose if he left the Chamber.

The acoustics of the Chamber were poor, making oratory impossible. Legislators could communicate only by messenger, and were provided with funds to hire as

[2]In translating the progress condition, I have rendered the Paxon word *μαδζδωριπΐσετ* as *majority of the legislators*. Alternative translations of this word have been proposed and are discussed in Section 2.2.

[3]In one tragic incident, legislator *Τωυεγ* developed irreversible amnesia after being hit on the head by a falling statue just outside the Chamber.

many messengers as they needed. A messenger could be counted on not to garble messages, but he might forget that he had already delivered a message, and deliver it again. Like the legislators they served, messengers devoted only part of their time to parliamentary duties. A messenger might leave the Chamber to conduct some business—perhaps taking a six-month voyage—before delivering a message. He might even leave forever, in which case the message would never be delivered.

Although legislators and messengers could enter and leave at any time, when inside the Chamber they devoted themselves to the business of Parliament. While they remained in the Chamber, messengers delivered messages in a timely fashion and legislators reacted promptly to any messages they received.

The official records of Paxos claim that legislators and messengers were scrupulously honest and strictly obeyed parliamentary protocol. Most scholars discount this as propaganda, intended to portray Paxos as morally superior to its eastern neighbors. Dishonesty, although rare, undoubtedly did occur. However, because it was never mentioned in official documents, we have little knowledge of how Parliament coped with dishonest legislators or messengers. What evidence has been uncovered is discussed in Section 3.3.5.

2 The Single-Decree Synod

The Paxon Parliament evolved from an earlier ceremonial Synod of priests that was convened every 19 years to choose a single, symbolic decree. For centuries, the Synod had chosen the decree by a conventional procedure that required all priests to be present. But as commerce flourished, priests began wandering in and out of the Chamber while the Synod was in progress. Finally, the old protocol failed, and a Synod ended with no decree chosen. To prevent a repetition of this theological disaster, Paxon religious leaders asked mathematicians to formulate a protocol for choosing the Synod's decree. The protocol's requirements and assumptions were essentially the same as those of the later Parliament except that instead of containing a sequence of decrees, a ledger would have at most one decree. The resulting Synod protocol is described here; the Parliamentary protocol is described in Section 3.

Mathematicians derived the Synod protocol in a series of steps. First, they proved results showing that a protocol satisfying certain constraints would guarantee consistency and allow progress. A *preliminary protocol* was then derived directly from these constraints. A restricted version of the preliminary protocol provided the *basic protocol* that guaranteed consistency, but not progress. The complete Synod protocol, satisfying the consistency and progress requirements, was obtained by restricting the basic protocol.[4]

The mathematical results are described in Section 2.1, and the protocols are described informally in Sections 2.2–2.4. A more formal description and correctness proof of the basic protocol appears in the appendix.

[4]The complete history of the Synod protocol's discovery is not known. Like modern computer scientists, Paxon mathematicians would describe elegant, logical derivations that bore no resemblance to how the algorithms were actually derived. However, it is known that the mathematical results (Theorems 1 and 2 of Section 2.1) really did precede the protocol. They were discovered when mathematicians, in response to the request for a protocol, were attempting to prove that a satisfactory protocol was impossible.

2.1 Mathematical Results

The Synod's decree was chosen through a series of numbered *ballots*, where a ballot was a referendum on a single decree. In each ballot, a priest had the choice only of voting for the decree or not voting.[5] Associated with a ballot was a set of priests called a *quorum*. A ballot succeeded iff (if and only if) every priest in the quorum voted for the decree. Formally, a ballot B consisted of the following four components. (Unless otherwise qualified, *set* is taken to mean *finite set*.[6])

B_{dec} A decree (the one being voted on).

B_{qrm} A nonempty set of priests (the ballot's quorum).

B_{vot} A set of priests (the ones who cast votes for the decree).[7]

B_{bal} A ballot number.

A ballot B was said to be *successful* iff $B_{qrm} \subseteq B_{vot}$, so a successful ballot was one in which every quorum member voted.

Ballot numbers were chosen from an unbounded ordered set of numbers. If $B'_{bal} > B_{bal}$, then ballot B' was said to be *later* than ballot B. However, this indicated nothing about the order in which ballots were conducted; a later ballot could actually have taken place before an earlier one.

Paxon mathematicians defined three conditions on a set $\mathcal{B}$ of ballots, and then showed that consistency was guaranteed and progress was possible if the set of ballots that had taken place satisfied those conditions. The first two conditions were simple; they can be stated informally as follows.

$B1(\mathcal{B})$ Each ballot in $\mathcal{B}$ has a unique ballot number.

$B2(\mathcal{B})$ The quorums of any two ballots in $\mathcal{B}$ have at least one priest in common.

The third condition was more complicated. One Paxon manuscript contained the following, rather confusing, statement of it.

$B3(\mathcal{B})$ For every ballot B in $\mathcal{B}$, if any priest in B's quorum voted in an earlier ballot in $\mathcal{B}$, then the decree of B equals the decree of the latest of those earlier ballots.

Interpretation of this cryptic text was aided by the manuscript pictured in Figure 1, which illustrates condition $B3(\mathcal{B})$ with a set $\mathcal{B}$ of five ballots for a Synod consisting of the five priests A, B, Γ, Δ, and E. This set $\mathcal{B}$ contains five ballots, where for each ballot, the set of voters is the subset of the priests in the quorum whose names are enclosed in boxes. For example, ballot number 14 has decree α, a quorum containing three priests, and a set of two voters. Condition $B3(\mathcal{B})$ has the form "for every B in $\mathcal{B}$: ...", where "..." is a condition on ballot B. The conditions for the five ballots B of Figure 1 are as follows.

[5] Like some modern nations, Paxos had not fully grasped the nature of Athenian democracy.

[6] Although Paxon mathematicians were remarkably advanced for their time, they obviously had no knowledge of set theory. I have taken the liberty of translating the Paxon's more primitive notation into the language of modern set theory.

[7] Only priests in the quorum actually voted, but Paxon mathematicians found it easier to convince people that the protocol was correct if, in their proof, they allowed any priest to vote in any ballot.

#	*decree*	*quorum and* $\boxed{\textit{voters}}$				
2	α	A	B	Γ	$\boxed{\Delta}$	
5	β	A	B	$\boxed{\Gamma}$		E
14	α		$\boxed{\text{B}}$		Δ	$\boxed{\text{E}}$
27	β	$\boxed{\text{A}}$		$\boxed{\Gamma}$	$\boxed{\Delta}$	
29	β		$\boxed{\text{B}}$	Γ	Δ	

Fig. 1. Paxon manuscript showing a set $\mathcal{B}$, consisting of five ballots, that satisfies conditions $B1(\mathcal{B})$–$B3(\mathcal{B})$. (Explanatory column headings have been added.)

2. Ballot number 2 is the earliest ballot, so the condition on that ballot is trivially true.
5. None of ballot 5's four quorum members voted in an earlier ballot, so the condition on ballot 5 is also trivially true.
14. The only member of ballot 14's quorum to vote in an earlier ballot is Δ, who voted in ballot number 2, so the condition requires that ballot 14's decree must equal ballot 2's decree.
27. (This is a successful ballot.) The members of ballot 27's quorum are A, Γ, and Δ. Priest A did not vote in an earlier ballot, the only earlier ballot Γ voted in was ballot 5, and the only earlier ballot Δ voted in was ballot 2. The latest of these two earlier ballots is ballot 5, so the condition requires that ballot 27's decree must equal ballot 5's decree.
29. The members of ballot 29's quorum are B, Γ, and Δ. The only earlier ballot that B voted in was number 14, priest Γ voted in ballots 5 and 27, and Δ voted in ballots 2 and 27. The latest of these four earlier ballots is number 27, so the condition requires that ballot 29's decree must equal ballot 27's decree.

To state $B1(\mathcal{B})$–$B3(\mathcal{B})$ formally requires some more notation. A *vote* v was defined to be a quantity consisting of three components: a priest v_{pst}, a ballot number v_{bal}, and a decree v_{dec}. It represents a vote cast by priest v_{pst} for decree v_{dec} in ballot number v_{bal}. The Paxons also defined *null* votes to be votes v with $v_{bal} = -\infty$ and $v_{dec} = \text{BLANK}$, where $-\infty < b < \infty$ for any ballot number b, and BLANK is not a decree. For any priest p, they defined $null_p$ to be the unique null vote v with $v_{pst} = p$.

Paxon mathematicians defined a total ordering on the set of all votes, but part of the manuscript containing the definition has been lost. The remaining fragment indicates that, for any votes v and v', if $v_{bal} < v'_{bal}$ then $v < v'$. It is not known how the relative order of v and v' was defined if $v_{bal} = v'_{bal}$.

For any set $\mathcal{B}$ of ballots, the set $Votes(\mathcal{B})$ of votes in $\mathcal{B}$ was defined to consist of all votes v such that $v_{pst} \in B_{vot}$, $v_{bal} = B_{bal}$, and $v_{dec} = B_{dec}$ for some $B \in \mathcal{B}$. If p is a priest and b is either a ballot number or $\pm\infty$, then $MaxVote(b, p, \mathcal{B})$ was

defined to be the largest vote v in $Votes(\mathcal{B})$ cast by p with $v_{bal} < b$, or to be $null_p$ if there was no such vote. Since $null_p$ is smaller than any real vote cast by p, this means that $MaxVote(b, p, \mathcal{B})$ is the largest vote in the set

$$\{v \in Votes(\mathcal{B}) : (v_{pst} = p) \wedge (v_{bal} < b)\} \cup \{null_p\}$$

For any nonempty set Q of priests, $MaxVote(b, Q, \mathcal{B})$ was defined to equal the maximum of all votes $MaxVote(b, p, \mathcal{B})$ with p in Q.

Conditions $B1(\mathcal{B})$–$B3(\mathcal{B})$ are stated formally as follows.[8]

$$\begin{aligned}
B1(\mathcal{B}) &\triangleq \forall B, B' \in \mathcal{B} : (B \neq B') \Rightarrow (B_{bal} \neq B'_{bal}) \\
B2(\mathcal{B}) &\triangleq \forall B, B' \in \mathcal{B} : B_{qrm} \cap B'_{qrm} \neq \emptyset \\
B3(\mathcal{B}) &\triangleq \forall B \in \mathcal{B} : (MaxVote(B_{bal}, B_{qrm}, \mathcal{B})_{bal} \neq -\infty) \Rightarrow \\
&\qquad (B_{dec} = MaxVote(B_{bal}, B_{qrm}, \mathcal{B})_{dec})
\end{aligned}$$

Although the definition of *MaxVote* depends upon the ordering of votes, $B1(\mathcal{B})$ implies that $MaxVote(b, Q, \mathcal{B})_{dec}$ is independent of how votes with equal ballot numbers were ordered.

To show that these conditions imply consistency, the Paxons first showed that $B1(\mathcal{B})$–$B3(\mathcal{B})$ imply that, if a ballot B in $\mathcal{B}$ is successful, then any later ballot in $\mathcal{B}$ is for the same decree as B.

Lemma *If $B1(\mathcal{B})$, $B2(\mathcal{B})$, and $B3(\mathcal{B})$ hold, then*

$$((B_{qrm} \subseteq B_{vot}) \wedge (B'_{bal} > B_{bal})) \Rightarrow (B'_{dec} = B_{dec})$$

for any B, B' in $\mathcal{B}$.

Proof of Lemma

For any ballot B in $\mathcal{B}$, let $\Psi(B, \mathcal{B})$ be the set of ballots in $\mathcal{B}$ later than B for a decree different from B's:

$$\Psi(B, \mathcal{B}) \triangleq \{B' \in \mathcal{B} : (B'_{bal} > B_{bal}) \wedge (B'_{dec} \neq B_{dec})\}$$

To prove the lemma, it suffices to show that if $B_{qrm} \subseteq B_{vot}$ then $\Psi(B, \mathcal{B})$ is empty. The Paxons gave a proof by contradiction. They assumed the existence of a B with $B_{qrm} \subseteq B_{vot}$ and $\Psi(B, \mathcal{B}) \neq \emptyset$, and obtained a contradiction as follows.[9]

1. Choose $C \in \Psi(B, \mathcal{B})$ such that $C_{bal} = \min\{B'_{bal} : B' \in \Psi(B, \mathcal{B})\}$.
   PROOF: C exists because $\Psi(B, \mathcal{B})$ is nonempty and finite.
2. $C_{bal} > B_{bal}$
   PROOF: By 1 and the definition of $\Psi(B, \mathcal{B})$.
3. $B_{vot} \cap C_{qrm} \neq \emptyset$
   PROOF: By $B2(\mathcal{B})$ and the hypothesis that $B_{qrm} \subseteq B_{vot}$.

---

[8] I use the Paxon mathematical symbol $\triangleq$, which meant *equals by definition*.

[9] Paxon mathematicians always provided careful, structured proofs of important theorems. They were not as sophisticated as modern mathematicians, who can omit many details and write paragraph-style proofs without ever making a mistake.

4. $MaxVote(C_{bal}, C_{qrm}, \mathcal{B})_{bal} \geq B_{bal}$
   PROOF: By 2, 3 and the definition of $MaxVote(C_{bal}, C_{qrm}, \mathcal{B})$.
5. $MaxVote(C_{bal}, C_{qrm}, \mathcal{B}) \in Votes(\mathcal{B})$
   PROOF: By 4 (which implies that $MaxVote(C_{bal}, C_{qrm}, \mathcal{B})$ is not a null vote) and the definition of $MaxVote(C_{bal}, C_{qrm}, \mathcal{B})$.
6. $MaxVote(C_{bal}, C_{qrm}, \mathcal{B})_{dec} = C_{dec}$.
   PROOF: By 5 and $B3(\mathcal{B})$.
7. $MaxVote(C_{bal}, C_{qrm}, \mathcal{B})_{dec} \neq B_{dec}$
   PROOF: By 6, 1, and the definition of $\Psi(B, \mathcal{B})$.
8. $MaxVote(C_{bal}, C_{qrm}, \mathcal{B})_{bal} > B_{bal}$
   PROOF: By 4, since 7 and $B1(\mathcal{B})$ imply that $MaxVote(C_{bal}, C_{qrm}, \mathcal{B})_{bal} \neq B_{bal}$.
9. $MaxVote(C_{bal}, C_{qrm}, \mathcal{B}) \in Votes(\Psi(B, \mathcal{B}))$
   PROOF: By 7, 8, and the definition of $\Psi(B, \mathcal{B})$.
10. $MaxVote(C_{bal}, C_{qrm}, \mathcal{B})_{bal} < C_{bal}$
    PROOF: By definition of $MaxVote(C_{bal}, C_{qrm}, \mathcal{B})$.
11. Contradiction
    PROOF: By 9, 10, and 1.

End Proof of Lemma

With this lemma, it was easy to show that, if $B1$–$B3$ hold, then any two successful ballots are for the same decree.

THEOREM 1. *If $B1(\mathcal{B})$, $B2(\mathcal{B})$, and $B3(\mathcal{B})$ hold, then*

$$((B_{qrm} \subseteq B_{vot}) \wedge (B'_{qrm} \subseteq B'_{vot})) \;\Rightarrow\; (B'_{dec} = B_{dec})$$

for any B, B' in $\mathcal{B}$.

Proof of Theorem
If $B'_{bal} = B_{bal}$, then $B1(\mathcal{B})$ implies $B' = B$. If $B'_{bal} \neq B_{bal}$, then the theorem follows immediately from the lemma.
End Proof of Theorem

The Paxons then proved a theorem asserting that if there are enough priests in the Chamber, then it is possible to conduct a successful ballot while preserving $B1$–$B3$. Although this does not guarantee progress, it at least shows that a balloting protocol based on $B1$–$B3$ will not deadlock.

THEOREM 2. *Let b be a ballot number and Q a set of priests such that $b > B_{bal}$ and $Q \cap B_{qrm} \neq \emptyset$ for all $B \in \mathcal{B}$. If $B1(\mathcal{B})$, $B2(\mathcal{B})$, and $B3(\mathcal{B})$ hold, then there is a ballot B' with $B'_{bal} = b$ and $B'_{qrm} = B'_{vot} = Q$ such that $B1(\mathcal{B} \cup \{B'\})$, $B2(\mathcal{B} \cup \{B'\})$, and $B3(\mathcal{B} \cup \{B'\})$ hold.*

Proof of Theorem
Condition $B1(\mathcal{B} \cup \{B'\})$ follows from $B1(\mathcal{B})$, the choice of B'_{bal}, and the assumption about b. Condition $B2(\mathcal{B} \cup \{B'\})$ follows from $B2(\mathcal{B})$, the choice of B'_{qrm}, and the assumption about Q. If $MaxVote(b, Q, \mathcal{B})_{bal} = -\infty$ then let B'_{dec} be any decree, else let it equal $MaxVote(b, Q, \mathcal{B})_{dec}$. Condition $B3(\mathcal{B} \cup \{B'\})$ then follows from $B3(\mathcal{B})$.
End Proof of Theorem

2.2 The Preliminary Protocol

The Paxons derived the *preliminary protocol* from the requirement that conditions $B1(\mathcal{B})$–$B3(\mathcal{B})$ remain true, where $\mathcal{B}$ was the set of all ballots that had been or were being conducted. The definition of the protocol specified how the set $\mathcal{B}$ changed, but the set was never explicitly calculated. The Paxons referred to $\mathcal{B}$ as a quantity observed only by the gods, since it might never be known to any mortal.

Each ballot was initiated by a priest, who chose its number, decree, and quorum. Each priest in the quorum then decided whether or not to vote in the ballot. The rules determining how the initiator chose a ballot's number, decree, and quorum, and how a priest decided whether or not to vote in a ballot were derived directly from the need to maintain $B1(\mathcal{B})$–$B3(\mathcal{B})$.

To maintain $B1$, each ballot had to receive a unique number. By remembering (with notes in his ledger) what ballots he had previously initiated, a priest could easily avoid initiating two different ballots with the same number. To keep different priests from initiating ballots with the same number, the set of possible ballot numbers was partitioned among the priests. While it is not known how this was done, an obvious method would have been to let a ballot number be a pair consisting of an integer and a priest, using a lexicographical ordering, where

$$(13,\ \Gamma\rho\alpha\check{\iota}) < (13,\ \Lambda\iota\nu\sigma\epsilon\check{\iota}) < (15,\ \Gamma\rho\alpha\check{\iota})$$

since Γ came before Λ in the Paxon alphabet. In any case, it is known that every priest had an unbounded set of ballot numbers reserved for his use.

To maintain $B2$, a ballot's quorum was chosen to contain a $\mu\alpha\delta\zeta\partial\omega\rho\iota\pi\check{\iota}\sigma\epsilon\tau$ of priests. Initially, $\mu\alpha\delta\zeta\partial\omega\rho\iota\pi\check{\iota}\sigma\epsilon\tau$ just meant a simple majority. Later, it was observed that fat priests were less mobile and spent more time in the Chamber than thin ones, so a $\mu\alpha\delta\zeta\partial\omega\rho\iota\pi\check{\iota}\sigma\epsilon\tau$ was taken to mean any set of priests whose total weight was more than half the total weight of all priests, rather than a simple majority of the priests. When a group of thin priests complained that this was unfair, actual weights were replaced with symbolic weights based on a priest's attendance record. The primary requirement for a $\mu\alpha\delta\zeta\partial\omega\rho\iota\pi\check{\iota}\sigma\epsilon\tau$ was that any two sets containing a $\mu\alpha\delta\zeta\partial\omega\rho\iota\pi\check{\iota}\sigma\epsilon\tau$ of priests had at least one priest in common. To maintain $B2$, the priest initiating a ballot B chose B_{qrm} to be a majority set.

Condition $B3$ requires that if $MaxVote(b,\ Q,\ \mathcal{B})_{dec}$ is not equal to BLANK, then a ballot with number b and quorum Q must have decree $MaxVote(b,\ Q,\ \mathcal{B})_{dec}$. If $MaxVote(b,\ Q,\ \mathcal{B})_{dec}$ equals BLANK, then the ballot can have any decree. To maintain $B3(\mathcal{B})$, before initiating a new ballot with ballot number b and quorum Q, a priest p had to find $MaxVote(b,\ Q,\ \mathcal{B})_{dec}$. To do this, p had to find $MaxVote(b,\ q,\ \mathcal{B})$ for each priest q in Q.

Recall that $MaxVote(b,\ q,\ \mathcal{B})$ is the vote with the largest ballot number less than b among all the votes cast by q, or $null_q$ if q did not vote in any ballot numbered less than b. Priest p obtains $MaxVote(b,\ q,\ \mathcal{B})$ from q by an exchange of messages. Therefore, the first two steps in the protocol for conducting a single ballot initiated by p are:[10]

[10]Priests p and q could be the same. For simplicity, the protocol is described with p sending messages to himself in this case. In reality, a priest could talk to himself without the use of messengers.

(1) Priest p chooses a new ballot number b and sends a *NextBallot*(b) message to some set of priests.

(2) A priest q responds to the receipt of a *NextBallot*(b) message by sending a *LastVote*(b, v) message to p, where v is the vote with the largest ballot number less than b that q has cast, or his null vote $null_q$ if q did not vote in any ballot numbered less than b.

Priest q must use notes in the back of his ledger to remember what votes he had previously cast.

When q sends the *LastVote*(b, v) message, v equals *MaxVote*(b, q, $\mathcal{B}$). But the set $\mathcal{B}$ of ballots changes as new ballots are initiated and votes are cast. Since priest p is going to use v as the value of *MaxVote*(b, q, $\mathcal{B}$) when choosing a decree, to keep $B3(\mathcal{B})$ true it is necessary that *MaxVote*(b, q, $\mathcal{B}$) not change after q has sent the *LastVote*(b, v) message. To keep *MaxVote*(b, q, $\mathcal{B}$) from changing, q must cast no new votes with ballot numbers between v_{bal} and b. By sending the *LastVote*(b, v) message, q is promising not to cast any such vote. (To keep this promise, q must record the necessary information in his ledger.)

The next two steps in the balloting protocol (begun in step 1 by priest p) are:

(3) After receiving a *LastVote*(b, v) message from every priest in some majority set Q, priest p initiates a new ballot with number b, quorum Q, and decree d, where d is chosen to satisfy $B3$. He then records the ballot in the back of his ledger and sends a *BeginBallot*(b, d) message to every priest in Q.

(4) Upon receipt of the *BeginBallot*(b, d) message, priest q decides whether or not to cast his vote in ballot number b. (He may not cast the vote if doing so would violate a promise implied by a *LastVote*(b', v') message he has sent for some other ballot.) If q decides to vote for ballot number b, then he sends a *Voted*(b, q) message to p and records the vote in the back of his ledger.

The execution of step 3 is considered to add a ballot B to $\mathcal{B}$, where $B_{bal} = b$, $B_{qrm} = Q$, $B_{vot} = \emptyset$ (no one has yet voted in this ballot), and $B_{dec} = d$. In step 4, if priest q decides to vote in the ballot, then executing that step is considered to change the set $\mathcal{B}$ of ballots by adding q to the set B_{vot} of voters in the ballot $B \in \mathcal{B}$.

A priest has the option not to vote in step 4, even if casting a vote would not violate any previous promise. In fact, all the steps in this protocol are optional. For example, a priest q can ignore a *NextBallot*(b) message instead of executing step 2. Failure to take an action can prevent progress, but it cannot cause any inconsistency because it cannot make $B1(\mathcal{B})$–$B3(\mathcal{B})$ false. Since the only effect not receiving a message can have is to prevent an action from happening, message loss also cannot cause inconsistency. Thus, the protocol guarantees consistency even if priests leave the chamber or messages are lost.

Receiving multiple copies of a message can cause an action to be repeated. Except in step 3, performing the action a second time has no effect. For example, sending several *Voted*(b, q) messages in step 4 has the same effect as sending just one. The repetition of step 3 is prevented by using the entry made in the back of the ledger when it is executed. Thus, the consistency condition is maintained even if a messenger delivers the same message several times.

Steps 1–4 describe the complete protocol for initiating a ballot and voting on it.

All that remains is to determine the results of the balloting and announce when a decree has been selected. Recall that a ballot is successful iff every priest in the quorum has voted. The decree of a successful ballot is the one chosen by the Synod. The rest of the protocol is:

(5) If p has received a *Voted*(b, q) message from every priest q in Q (the quorum for ballot number b), then he writes d (the decree of that ballot) in his ledger and sends a *Success*(d) message to every priest.

(6) Upon receiving a *Success*(d) message, a priest enters decree d in his ledger.

Steps 1–6 describe how an individual ballot is conducted. The preliminary protocol allows any priest to initiate a new ballot at any time. Each step maintains $B1(\mathcal{B})$–$B3(\mathcal{B})$, so the entire protocol also maintains these conditions. Since a priest enters a decree in his ledger only if it is the decree of a successful ballot, Theorem 1 implies that the priests' ledgers are consistent. The protocol does not address the question of progress.

In step 3, if the decree d is determined by condition $B3$, then it is possible that this decree is already written in the ledger of some priest. That priest need not be in the quorum Q; he could have left the Chamber. Thus, consistency would not be guaranteed if step 3 allowed any greater freedom in choosing d.

2.3 The Basic Protocol

In the preliminary protocol, a priest must record (i) the number of every ballot he has initiated, (ii) every vote he has cast, and (iii) every *LastVote* message he has sent. Keeping track of all this information would have been difficult for the busy priests. The Paxons therefore restricted the preliminary protocol to obtain the more practical *basic protocol* in which each priest p had to maintain only the following information in the back of his ledger:

lastTried$[p]$ The number of the last ballot that p tried to initiate, or $-\infty$ if there was none.

prevVote$[p]$ The vote cast by p in the highest-numbered ballot in which he voted, or $-\infty$ if he never voted.

nextBal$[p]$ The largest value of b for which p has sent a *LastVote*(b, v) message, or $-\infty$ if he has never sent such a message.

Steps 1–6 of the preliminary protocol describe how a single ballot is conducted by its initiator, priest p. The preliminary protocol allows p to conduct any number of ballots concurrently. In the basic protocol, he conducts only one ballot at a time—ballot number *lastTried*$[p]$. After p initiates this ballot, he ignores messages that pertain to any other ballot that he had previously initiated. Priest p keeps all information about the progress of ballot number *lastTried*$[p]$ on a slip of paper. If he loses that slip of paper, then he stops conducting the ballot.

In the preliminary protocol, each *LastVote*(b, v) message sent by a priest q represents a promise not to vote in any ballot numbered between v_{bal} and b. In the basic protocol, it represents the stronger promise not to cast a new vote in any ballot numbered less than b. This stronger promise might prevent him from casting a vote in step 4 of the basic protocol that he would have been allowed to cast in the preliminary protocol. However, since the preliminary protocol always gives q

the option of not casting his vote, the basic protocol does not require him to do anything not allowed by the preliminary protocol.

Steps 1–6 of the preliminary protocol become the following six steps for conducting a ballot in the basic protocol. (All information used by p to conduct the ballot, other than $lastTried[p]$, $prevVote[p]$, and $nextBal[p]$, is kept on a slip of paper.)

(1) Priest p chooses a new ballot number b greater than $lastTried[p]$, sets $lastTried[p]$ to b, and sends a $NextBallot(b)$ message to some set of priests.

(2) Upon receipt of a $NextBallot(b)$ message from p with $b > nextBal[q]$, priest q sets $nextBal[q]$ to b and sends a $LastVote(b, v)$ message to p, where v equals $prevVote[q]$. (A $NextBallot(b)$ message is ignored if $b \leq nextBal[q]$.)

(3) After receiving a $LastVote(b, v)$ message from every priest in some majority set Q, where $b = lastTried[p]$, priest p initiates a new ballot with number b, quorum Q, and decree d, where d is chosen to satisfy $B3$. He then sends a $BeginBallot(b, d)$ message to every priest in Q.

(4) Upon receipt of a $BeginBallot(b, d)$ message with $b = nextBal[q]$, priest q casts his vote in ballot number b, sets $prevVote[q]$ to this vote, and sends a $Voted(b, q)$ message to p. (A $BeginBallot(b, d)$ message is ignored if $b \neq nextBal[q]$.)

(5) If p has received a $Voted(b, q)$ message from every priest q in Q (the quorum for ballot number b), where $b = lastTried[p]$, then he writes d (the decree of that ballot) in his ledger and sends a $Success(d)$ message to every priest.

(6) Upon receiving a $Success(d)$ message, a priest enters decree d in his ledger.

The basic protocol is a restricted version of the preliminary protocol, meaning that every action allowed by the basic protocol is also allowed by the preliminary protocol. Since the preliminary protocol satisfies the consistency condition, the basic protocol also satisfies that condition. Like the preliminary protocol, the basic protocol does not require that any action ever be taken, so it does not addresses the question of progress.

The derivation of the basic protocol from $B1$–$B3$ made it obvious that the consistency condition was satisfied. However, some similarly "obvious" ancient wisdom had turned out to be false, and skeptical citizens demanded a more rigorous proof. Their Paxon mathematicians' proof that the protocol satisfies the consistency condition is reproduced in the appendix.

2.4 The Complete Synod Protocol

The basic protocol maintains consistency, but it cannot ensure any progress because it states only what a priest may do; it does not require him to do anything. The complete protocol consists of the same six steps for conducting a ballot as the basic protocol. To help achieve progress, it includes the obvious additional requirement that priests perform steps 2–6 of the protocol as soon as possible. However, to meet the progress condition, it is necessary that some priest be required to perform step 1, which initiates a ballot. The key to the complete protocol lay in determining when a priest should initiate a ballot.

Never initiating a ballot will certainly prevent progress. However, initiating too may ballots can also prevent progress. If b is larger than any other ballot number, then the receipt of a $NextBallot(b)$ message by priest q in step 2 may elicit a

promise that prevents him from voting in step 4 for any previously initiated ballot. Thus, the initiation of a new ballot can prevent any previously initiated ballot from succeeding. If new ballots are continually initiated with increasing ballot numbers before the previous ballots have a chance to succeed, then no progress might be made.

Achieving the progress condition requires that new ballots be initiated until one succeeds, but that they not be initiated too frequently. To develop the complete protocol, the Paxons first had to know how long it took messengers to deliver messages and priests to respond. They determined that a messenger who did not leave the Chamber would always deliver a message within 4 minutes, and a priest who remained in the Chamber would always perform an action within 7 minutes of the event that caused the action.[11] Thus, if p and q were in the Chamber when some event caused p to send a message to q, and q responded with a reply to p, then p would receive that reply within 22 minutes if neither messenger left the Chamber. (Priest p would send the message within 7 minutes of the event, q would receive the message within 4 more minutes, he would respond within 7 minutes, and the reply would reach p within 4 more minutes.)

Suppose that only a single priest p was initiating ballots, and that he did so by sending a message to every priest in step 1 of the protocol. If p initiated a ballot when a majority set of priests was in the chamber, then he could expect to execute step 3 within 22 minutes of initiating the ballot, and to execute step 5 within another 22 minutes. If he was unable to execute the steps by those times, then either some priest or messenger left the Chamber after p initiated the ballot, or a larger-numbered ballot had previously been initiated by another priest (before p became the only priest to initiate ballots). To handle the latter possibility, p had to learn about any ballot numbers greater than $lastTried[p]$ used by other priests. This could be done by extending the protocol to require that if a priest q received a $NextBallot(b)$ or a $BeginBallot(b, d)$ message from p with $b < nextBal[q]$, then he sent p a message containing $nextBal[q]$. Priest p would then initiate a new ballot with a larger ballot number.

Still assuming that p was the only priest initiating ballots, suppose that he were required to initiate a new ballot iff (i) he had not executed step 3 or step 5 within the previous 22 minutes, or (ii) he learned that another priest had initiated a higher-numbered ballot. If the Chamber doors were locked with p and a majority set of priests inside, then a decree would be passed and recorded in the ledgers of all priests in the Chamber within 99 minutes. (It could take 22 minutes for p to start the next ballot, 22 more minutes to learn that another priest had initiated a larger-numbered ballot, then 55 minutes to complete steps 1–6 for a successful ballot.) Thus, the progress condition would be met if only a single priest, who did not leave the chamber, were initiating ballots.

The complete protocol therefore included a procedure for choosing a single priest, called the *president*, to initiate ballots. In most forms of government, choosing a

[11] I am assuming a value of 30 seconds for the $\delta\zeta\partial\iota\phi\check{\iota}$, the Paxon unit of time. This value is within the range determined from studies of hourglass shards. The reaction time of priests was so long because they had to respond to every message within 7 minutes (14 $\delta\zeta\partial\iota\phi\check{\iota}$), even if a number of messages arrived simultaneously.

president can be a difficult problem. However, the difficultly arises only because most governments require that there be exactly one president at any time. In the United States, for example, chaos would result if some people thought Bush had been elected president while others thought that Dukakis had, since one of them might decide to sign a bill into law while the other decided to veto it. However, in the Paxon Synod, having multiple presidents could only impede progress; it could not cause inconsistency. For the complete protocol to satisfy the progress condition, the method for choosing the president needed only to satisfy the following *presidential selection requirement*:

> If no one entered or left the Chamber, then after T minutes exactly one priest in the Chamber would consider himself to be the president.

If the presidential selection requirement were met, then the complete protocol would have the property that if a majority set of priests were in the chamber and no one entered or left the Chamber for $T + 99$ minutes, then at the end of that period every priest in the Chamber would have a decree written in his ledger.

The Paxons chose as president the priest whose name was last in alphabetical order among the names of all priests in the Chamber, though we don't know exactly how this was done. The presidential selection requirement would have been satisfied if a priest in the Chamber sent a message containing his name to every other priest at least once every $T - 11$ minutes, and a priest considered himself to be president iff he received no message from a "higher-named" priest for T minutes.

The complete Synod protocol was obtained from the basic protocol by requiring priests to perform steps 2–6 promptly, adding a method for choosing a president who initiated ballots, and requiring the president to initiate ballots at the appropriate times. Many details of the protocol are not known. I have described simple methods for selecting a president and for deciding when the president should initiate a new ballot, but they are undoubtedly not the ones used in Paxos. The rules I have given require the president to keep initiating ballots even after a decree has been chosen, thereby ensuring that priests who have just entered the Chamber learn about the chosen decree. There were obviously better ways to make sure priests learned about the decree after it had been chosen. Also, in the course of selecting a president, each priest probably sent his value of $lastTried[p]$ to the other priests, allowing the president to choose a large enough ballot number on his first try.

The Paxons realized that any protocol to achieve the progress condition must involve measuring the passage of time.[12] The protocols given above for selecting a president and initiating ballots are easily formulated as precise algorithms that set timers and perform actions when time-outs occur—assuming perfectly accurate timers. A closer analysis reveals that such protocols can be made to work with timers having a known bound on their accuracy. The skilled glass blowers of Paxos had no difficulty constructing suitable hourglass timers.

Given the sophistication of Paxon mathematicians, it is widely believed that they must have found an optimal algorithm to satisfy the presidential selection requirement. We can only hope that this algorithm will be discovered in future

[12]However, many centuries were to pass before a rigorous proof of this result was given.[Fischer et al. 1985]

excavations on Paxos.

3 The Multi-Decree Parliament

When Parliament was established, a protocol to satisfy its consistency and progress requirements was derived from the Synod protocol. The derivation and properties of the original parliamentary protocol are described in Sections 3.1 and 3.2. Section 3.3 discusses the further evolution of the protocol.

3.1 The Protocol

Instead of passing just one decree, the Paxon Parliament had to pass a series of numbered decrees. As in the Synod protocol, a president was elected. Anyone who wanted a decree passed would inform the president, who would assign a number to the decree and attempt to pass it. Logically, the parliamentary protocol used a separate instance of the complete Synod protocol for each decree number. However, a single president was selected for all these instances, and he performed the first two steps of the protocol just once.

The key to deriving the parliamentary protocol is the observation that, in the Synod protocol, the president does not choose the decree or the quorum until step 3. A newly elected president p can send to some set of legislators a single message that serves as the $NextBallot(b)$ message for all instances of the Synod protocol. (There are an infinite number of instances—one for each decree number.) A legislator q can reply with a single message that serves as the *LastVote* messages for step 2 of all instances of the Synod protocol. This message contains only a finite amount of information, since q can have voted in only a finite number of instances.

When the new president has received a reply from every member of a majority set, he is ready to perform step 3 for every instance of the Synod protocol. For some finite number of instances (decree numbers), the choice of decree in step 3 will be determined by $B3$. The president immediately performs step 3 for each of those instances to try passing these decrees. Then, whenever he receives a request to pass a decree, he chooses the lowest-numbered decree that he is still free to choose, and he performs step 3 for that decree number (instance of the Synod protocol) to try to pass the decree.

The following modifications to this simple protocol lead to the actual Paxon Parliament's protocol.

—There is no reason to go through the Synod protocol for a decree number whose outcome is already known. Therefore, if a newly elected president p has all decrees with numbers less than or equal to n written in his ledger, then he sends a $NextBallot(b, n)$ message that serves as a $NextBallot(b)$ message in all instances of the Synod protocol for decree numbers larger than n. In his response to this message, legislator q informs p of all decrees numbered greater than n that already appear in q's ledger (in addition to sending the usual *LastVote* information for decrees not in his ledger), and he asks p to send him any decrees numbered n or less that are not in his ledger.

—Suppose decrees 125 and 126 are introduced late Friday afternoon, decree 126 is passed and is written in one or two ledgers, but before anything else happens, the legislators all go home for the weekend. Suppose also that the following Monday,

$\Delta\phi\omega\rho\kappa$ is elected the new president and learns about decree 126, but she has no knowledge of decree 125 because the previous president and all legislators who had voted for it are still out of the Chamber. She will hold a ballot that passes decree 126, which leaves a gap in the ledgers. Assigning number 125 to a new decree would cause it to appear earlier in the ledger than decree 126, which had been passed the previous week. Passing decrees out of order in this way might cause confusion—for example, if the citizen who proposed the new decree did so because he knew decree 126 had already passed. Instead, $\Delta\phi\omega\rho\kappa$ would attempt to pass

125: *The ides of February is national olive day*

a traditional decree that made absolutely no difference to anyone in Paxos. In general, a new president would fill any gaps in his ledger by passing the "olive-day" decree.

The consistency and progress properties of the parliamentary protocol follow immediately from the corresponding properties of the Synod protocol from which it was derived. To our knowledge, the Paxons never bothered writing a precise description of the parliamentary protocol because it was so easily derived from the Synod protocol.

3.2 Properties of the Protocol

3.2.1 *The Ordering of Decrees* Balloting could take place concurrently for many different decree numbers, with ballots initiated by different legislators—each thinking he was president when he initiated the ballot. We cannot say precisely in what order decrees would be passed, especially without knowing how a president was selected. However, there is one important property about the ordering of decrees that can be deduced.

A decree was said to to be *proposed* when it was chosen by the president in step 3 of the corresponding instance of the Synod protocol. The decree was said to be *passed* when it was written for the first time in a ledger. Before a president could propose any new decrees, he had to learn from all the members of a majority set what decrees they had voted for. Any decree that had already been passed must have been voted for by at least one legislator in the majority set. Therefore, the president must have learned about all previously passed decrees before initiating any new decree. The president would not fill a gap in the ledgers with an important decree—that is, with any decree other than the "olive-day" decree. He would also not propose decrees out of order. Therefore, the protocol satisfied the following *decree-ordering property*.

> If decrees A and B are important and decree A was passed before decree B was proposed, then A has a lower decree number than B.

3.2.2 *Behind Closed Doors* Although we don't know the details involved in choosing a new president, we do know exactly how Parliament functioned when the president had been chosen and no one was entering or leaving the Chamber. Upon receiving a request to pass a decree—either directly from a citizen or relayed from another legislator—the president assigned the decree a number and passed it with the following exchange of messages. (The numbers refer to the corresponding steps in the Synod protocol.)

(3) The president sent a *BeginBallot* message to each legislator in a quorum.
(4) Each legislator in the quorum sent a *Voted* message to the president.
(5) The president sent a *Success* message to every legislator.

This is a total of three message delays and about $3N$ messages, assuming a parliament of N legislators and a quorum of about $N/2$. Moreover, if Parliament was busy, the president would combine the *BeginBallot* message for one decree with the *Success* message for a previous one, for a total of only $2N$ messages per decree.

3.3 Further Developments

Governing the island turned out to be a more complex task than the Paxons realized. A number of problems arose whose solutions required changes to the protocol. The most important of these changes are described below.

3.3.1 *Picking a President* The president of parliament was originally chosen by the method that had been used in the Synod, which was based purely on the alphabetical ordering of names. Thus, when legislator $\Omega\kappa\iota$ returned from a six-month vacation, he was immediately made president—even though he had no idea what had happened in his absence. Parliamentary activity came to a halt while $\Omega\kappa\iota$, who was a slow writer, laboriously copied six months worth of decrees to bring his ledger up to date.

This incident led to a debate about the best way to choose a president. Some Paxons urged that once a legislator became president, he should remain president until he left the Chamber. An influential group of citizens wanted the richest legislator in the Chamber to be president, since he could afford to hire more scribes and other servants to help him with the presidential duties. They argued that once a rich legislator had brought his ledger up to date, there was no reason for him not to assume the presidency. Others, however, argued that the most upstanding citizen should be made president, regardless of wealth. Upstanding probably meant less likely to be dishonest, although no Paxon would publicly admit the possibility of official malfeasance. Unfortunately, the outcome of this debate is not known; no record exists of the presidential selection protocol that was ultimately used.

3.3.2 *Long Ledgers* As the years progressed and Parliament passed more and more decrees, Paxons had to pore over an ever longer list of decrees to find the current olive tax or what color goat could be sold. A legislator who returned to the Chamber after an extended voyage had to do quite a bit of copying to bring his ledger up to date. Eventually, the legislators were forced to convert their ledgers from lists of decrees into law books that contained only the current state of the law and the number of the last decree whose passage was reflected in that state.

To learn the current olive tax, one looked in the law book under "taxes"; to learn what color goat could be sold, one looked under "mercantile law". If a legislator's ledger contained the law through decree 1298 and he learned that decree 1299 set the olive tax to 6 drachmas per ton, he just changed the entry for the olive-tax law and noted that his ledger was complete through decree 1299. If he then learned about decree 1302, he would write it down in the back of the ledger and wait until he learned about decrees 1300 and 1301 before incorporating decree 1302 into the law book.

To enable a legislator who had been gone for a short time to catch up without copying the entire law book, legislators kept a list of the past week's decrees in the back of the book. They could have kept this list on a slip of paper, but it was convenient for a legislator to enter decrees in the back of the ledger as they were passed and update the law book only two or three times a week.

3.3.3 *Bureaucrats* As Paxos prospered, legislators became very busy. Parliament could no longer handle all details of government, so a bureaucracy was established. Instead of passing a decree to declare whether each lot of cheese was fit for sale, Parliament passed a decree appointing a cheese inspector to make those decisions.

It soon became evident that selecting bureaucrats was not as simple as it first seemed. Parliament passed a decree making *Δĭκστρα* the first cheese inspector. After some months, merchants complained that *Δĭκστρα* was too strict and was rejecting perfectly good cheese. Parliament then replaced him by passing the decree

1375: *Γωυδα is the new cheese inspector*

But *Δĭκστρα* did not pay close attention to what Parliament did, so he did not learn of this decree right away. There was a period of confusion in the cheese market when both *Δĭκστρα* and *Γωυδα* were inspecting cheese and making conflicting decisions.

To prevent such confusion, the Paxons had to guarantee that a position could be held by at most one bureaucrat at any time. To do this, a president included as part of each decree the time and date when it was proposed. A decree making *Δĭκστρα* the cheese inspector might read

2716: *8:30 15 Jan 72—Δĭκστρα is cheese inspector for 3 months*

This declares his term to begin either at 8:30 on 15 January or when the previous inspector's term ended—whichever was later. His term would end at 8:30 on 15 March, unless he explicitly resigned by asking the president to pass a decree like

2834: *9:15 3 Mar 72—Δĭκστρα resigns as cheese inspector*

A bureaucrat was appointed for a short term, so he could be replaced quickly—for example, if he left the island. Parliament would pass a decree to extend the bureaucrat's term if he was doing a satisfactory job.

A bureaucrat needed to tell time to determine if he currently held a post. Mechanical clocks were unknown on Paxos, but Paxons could tell time accurately to within 15 minutes by the position of the sun or the stars.[13] If *Δĭκστρα*'s term began at 8:30, he would not start inspecting cheese until his celestial observations indicated that it was 8:45.

It is easy to make this method of appointing bureaucrats work if higher-numbered decrees always have later proposal times. But what if Parliament passed the decrees

2854: *9:45 9 Apr 78—Φρανσεζ is wine taster for 2 months*

2855: *9:20 9 Apr 78—Πνυελĭ is wine taster for 1 month*

[13]Cloudy days are rare in Paxos's balmy climate.

that were proposed between 9:30 and 9:35 by different legislators who both thought they were president? Such out-of-order proposal times are easily prevented because the parliamentary protocol satisfies the following property.

> If two decrees are passed by different presidents, then one of the presidents proposed his decree after learning that the other decree had been proposed.

To see that this property is satisfied, suppose that ballot number b was successful for decree D, ballot number b' was successful for decree D', and $b < b'$. Let q be a legislator who voted in both ballots. The balloting for D' began with a $NextBallot(b', n)$ message. If the sender of that message did not already know about D, then n is less than the decree number of D, and q's reply to the *NextBallot* message must state that he voted for D.

3.3.4 *Learning the Law* In addition to requesting the passage of decrees, ordinary citizens needed to inquire about the current law of the land. The Paxons at first thought that a citizen could simply examine the ledger of any legislator, but the following incident demonstrated that a more sophisticated approach was needed. For centuries, it had been legal to sell only white goats. A farmer named *Δωλεφ* got Parliament to pass the decree

77: *The sale of black goats is permitted*

Δωλεφ then instructed his goatherd to sell some black goats to a merchant named *Σκεεν*. As a law-abiding citizen, *Σκεεν* asked legislator *Σπωκμεĭρ* if such a sale would be legal. But *Σπωκμεĭρ* had been out of the Chamber and had no entry in his ledger past decree 76. He advised *Σκεεν* that the sale would be illegal under the current law, so *Σκεεν* refused to buy the goats.

This incident led to the formulation of the following *monotonicity condition* on inquiries about the law.

> If one inquiry precedes a second inquiry, then the second inquiry cannot reveal an earlier state of the law than the first.

If a citizen learns that a particular decree has been passed, then the process of acquiring that knowledge is considered to be an implicit inquiry to which this condition applies. As we will see, the interpretation of the monotonicity condition changed over the years.

Initially, the monotonicity condition was achieved by passing a decree for each inquiry. If *Σδνĭδερ* wanted to know the current tax on olives, he would get Parliament to pass a decree such as

87: *Citizen Σδνĭδερ is reading the law*

He would then read any ledger complete at least through decree 86 to learn the olive tax as of that decree. If citizen *Γρεες* then inquired about the olive tax, the decree for his inquiry was proposed after decree 87 was passed, so the decree-ordering property (Section 3.2.1) implies that it received a decree number greater than 87. Therefore, *Γρεες* could not obtain an earlier value of the olive tax than *Σδνĭδερ*.

This method of reading the law satisfied the monotonicity condition when *precedes* was interpreted to mean that inquiry A precedes inquiry B iff A finished at an earlier time than B began.

Passing a decree for every inquiry soon proved too cumbersome. The Paxons realized that a simpler method was possible if they weakened the monotonicity condition by changing the interpretation of *precedes*. They decided that for one event to precede another, the first event not only had to happen at an earlier time, but it had to be able to causally affect the second event. The weaker monotonicity condition prevents the problem first encountered by farmer *Δωλεφ* and merchant *Σκεεν* because there is a causal chain of events between the end of the implicit inquiry by *Δωλεφ* and the beginning of the inquiry by *Σκεεν*.

The weaker monotonicity condition was met by using decree numbers in all business transactions and inquiries. For example, farmer *Δωλεφ*, whose flock included many nonwhite goats, got Parliament to pass the decree

277: *The sale of brown goats is permitted*

When selling his brown goats to *Σκεεν*, he informed the merchant that the sale was legal as of decree number 277. *Σκεεν* then asked legislator *Σπωκμεĭρ* if the sale were legal under the law through at least decree 277. If *Σπωκμεĭρ*'s ledger was not complete through decree 277, he would either wait until it was or else tell *Σκεεν* to ask someone else. If *Σπωκμεĭρ*'s ledger went through decree 298, then he would tell *Σκεεν* that the sale was legal as of decree number 298. Merchant *Σκεεν* would remember the number 298 for use in his next business transaction or inquiry about the law.

The Paxons had satisfied the monotonicity condition, but ordinary citizens disliked having to remember decree numbers. Again, the Paxons solved the problem by re-interpreting the monotonicity condition—this time, by changing the meaning of *state of the law*. They divided the law into separate areas, and a legislator was chosen as specialist for each area. The current state of each area of the law was determined by that specialist's ledger. For example, suppose decree 1517 changed the tariff law and decree 1518 changed the tax law. The tax law would change first if the tax-law specialist learned of both decrees before the tariff-law specialist learned of either, yielding a state of the law that could not be obtained by enacting the decrees in numerical order.

To avoid conflicting definitions of the current state, the Paxons required that there be at most one specialist at a time for any area. This requirement was satisfied by using the same method to choose specialists that was used to choose bureaucrats (see Section 3.3.3). If each inquiry involved only a single area of the law, monotonicity was then achieved by directing the inquiry to that area's specialist, who answered it from his ledger. Since learning that a law had passed constituted the result of an implicit inquiry, the Paxons required that a decree change at most one area of the law, and that notification of the decree's passage could come only from the area's specialist.

Inquiries involving multiple areas were not hard to handle. When merchant *Λισκωφ* asked if the tariff on an imported golden fleece was higher than the sales tax on one purchased locally, the tax-law and tariff-law specialists had to cooperate to provide an answer. For example, the tax specialist could answer *Λισκωφ* by first

asking the tariff specialist for the tariff on golden fleeces, so long as he made no changes to his ledger before receiving a reply.

This method proved satisfactory until it became necessary to make a sweeping change to several areas of the law at one time. The Paxons then realized that the necessary requirement for maintaining monotonicity was not that a decree affect only a single area, but that every area it affects have the same specialist. Parliament could change several areas of the law with a single decree by first appointing a single legislator to be the specialist for all those areas. Moreover, the same area could have multiple specialists, so long as that area of the law was not allowed to change. Just before income taxes were due, Parliament would appoint several tax-law specialists to handle the seasonal flood of inquiries about the tax law.

3.3.5 *Dishonest Legislators and Honest Mistakes* Despite official assertions to the contrary, there must have been a few dishonest legislators in the history of Paxos. When caught, they were probably exiled. By sending contradictory messages, a malicious legislator could cause different legislators' ledgers to be inconsistent. Inconsistency could also result from a lapse of memory by an honest legislator or messenger.

When inconsistencies were recognized, they could easily be corrected by passing decrees. For example, disagreement about the current olive tax could be eliminated by passing a new decree declaring the tax to have a certain value. The difficult problem lay in correcting inconsistent ledgers even if no one was aware of the inconsistency.

The existence of dishonesty or mistakes by legislators can be inferred from the redundant decrees that began appearing in ledgers several years after the founding of Parliament. For example, the decree

2605: *The olive tax is 9 drachmas per ton*

was passed even though decree 2155 had already set the olive tax to 9 drachmas per ton, and no intervening decree had changed it. Parliament apparently cycled through its laws every six months so that even if legislators' ledgers were initially inconsistent, all legislators would agree on the current law of the land within six months. It is believed that by the use of these redundant decrees, the Paxons made their Parliament self-stabilizing. (*Self-stabilizing* is a modern term due to Dijkstra [Dijkstra 1974].)

It is not clear precisely what self-stabilization meant in a Parliament with legislators coming and going at will. The Paxons would not have been satisfied with a definition that required all legislators to be in the Chamber at one time before consistency could be guaranteed. However, achieving consistency required that if one legislator had an entry in his ledger for a certain decree number and another did not, then the second legislator would eventually fill in that entry.

Unfortunately, we don't know exactly what sort of self-stabilization property the Paxon Parliament possessed or how it was achieved. Paxon mathematicians undoubtedly addressed the problem, but their work has not yet been found. I hope that future archaeological expeditions to Paxos will give high priority to the search for manuscripts on self-stabilization.

3.3.6 *Choosing New Legislators* At first, membership in Parliament was hereditary, passing from parent to child. When the elder statesman *Παρνας* retired, he gave his ledger to his son, who carried on without interruption. It made no difference to other legislators which *Παρνας* they communicated with.

As old families emigrated and new ones immigrated, this system had to change. The Paxons decided to add and remove members of Parliament by decree. This posed a circularity problem: membership in Parliament was determined by which decrees were passed, but passing a decree required knowing what constituted a majority set, which in turn depended upon who was a member of Parliament. The circularity was broken by letting the membership of Parliament used in passing decree n be specified by the law as of decree $n - 3$. A president could not try to pass decree 3255 until he knew all decrees through decree 3252. In practice, after passing the decree

3252: *Στρωνγ is now a legislator*

the president would immediately pass the "olive-day" decree as decrees 3253 and 3254.

Changing the composition of Parliament in this way was dangerous and had to be done with care. The consistency and progress conditions would always hold. However, the progress condition guaranteed progress only if a majority set was in the Chamber; it did not guarantee that a majority set would ever be there. In fact, the mechanism for choosing legislators led to the downfall of the Parliamentary system in Paxos. Because of a scribe's error, a decree that was supposed to honor sailors who had drowned in a shipwreck instead declared them to be the only members of Parliament. Its passage prevented any new decrees from being passed—including the decrees proposed to correct the mistake. Government in Paxos came to a halt. A general named *Λαμπσων* took advantage of the confusion to stage a coup, establishing a military dictatorship that ended centuries of progressive government. Paxos grew weak under a series of corrupt dictators, and was unable to repel an invasion from the east that led to the destruction of its civilization.

4 Relevance to Computer Science

4.1 The State Machine Approach

Although Paxos's Parliament was destroyed many centuries ago, its protocol is still useful. For example, consider a simple distributed database system that might be used as a name server. A state of the database consists of an assignment of values to names. Copies of the database are maintained by multiple servers. A client program can issue, to any server, a request to read or change the value assigned to a name. There are two kinds of read request: a *slow read*, which returns the value currently assigned to a name, and a *fast read*, which is faster but might not reflect a recent change to the database.

There is an obvious correspondence between this database system and the Paxon Parliament:

Parliament		Distributed Database
legislator	$\leftrightarrow$	server
citizen	$\leftrightarrow$	client program
current law	$\leftrightarrow$	database state

command:	**read**(*name*, *client*)	**update**(*name*, *val*, *client*)
response:	(*client*, value of *name*)	(*client*, "ok")
new state:	Same as current state	Same as current state except value of *name* changed to *val*

Fig. 2. State machine for simple database.

A client's request to change a value is performed by passing a decree. A *slow read* involves passing a decree, as described in Section 3.3.4. A *fast read* is performed by reading the server's current version of the database. The Paxon Parliament protocol provides a distributed, fault-tolerant implementation of the database system,

This method of implementing a distributed database is an instance of the state machine approach, first proposed in [Lamport 1978]. In this approach, one first defines a *state machine*, which consists of a set of states, a set of commands, a set of responses, and a function that assigns a response/state pair (a pair consisting of a response and a state) to each command/state pair. Intuitively, a state machine executes a command by producing a response and changing its state; the command and the machine's current state determine its response and its new state. For the distributed database, a state-machine state is just a database state. The state-machine commands and the function specifying the response and new state are described in Figure 2.

In the state-machine approach, a system is implemented with a network of server processes. The servers transform client requests into state machine commands, execute the commands, and transform the state-machine responses into replies to clients. A general algorithm ensures that all servers obtain the same sequence of commands, thereby ensuring that they all produce the same sequence of responses and state changes—assuming they all start from the same initial state. In the database example, a client request to perform a *slow read* or to change a value is transformed into a state-machine **read** or **update** command. That command is executed, and the state-machine response is transformed into a reply to the client, which is sent to him by the server who received his request. Since all servers perform the same sequence of state-machine commands, they all maintain consistent versions of the database. However, at any time, some servers may have earlier versions than others because a state-machine command need not be executed at the same time by all servers. A server uses his current version of the state to reply to a *fast read* request, without executing a state-machine command.

The functionality of the system is expressed by the state machine, which is just a function from command/state pairs to response/state pairs. Problems of synchronization and fault-tolerance are handled by the general algorithm with which servers obtain the sequence of commands. When designing a new system, only the state machine is new. The servers obtain the state-machine commands by a standard distributed algorithm that has already been proved correct. Functions are much easier to design, and to get right, than distributed algorithms.

The first algorithm for implementing an arbitrary state machine appeared in [Lamport 1978]. Later, algorithms were devised to tolerate up to any fixed number f of arbitrary failures [Lamport 1984]. These algorithms guarantee that, if fewer

than f processes fail, then state machine commands are executed within a fixed length of time. The algorithms are thus suitable for applications requiring real-time response.[14] But if more than f failures occur, then different servers may have inconsistent copies of the state machine. Moreover, the inability of two servers to communicate with each other is equivalent to the failure of one of them. For a system to have a low probability of losing consistency, it must use an algorithm with a large value of f, which in turn implies a large cost in redundant hardware, communication bandwidth, and response time.

The Paxon Parliament's protocol provides another way to implement an arbitrary state machine. The legislators' law book corresponds to the machine state, and passing a decree corresponds to executing a state-machine command. The resulting algorithm is less robust and less expensive than the earlier algorithms. It does not tolerate arbitrary, malicious failures, nor does it guarantee bounded-time response. However, consistency is maintained despite the (benign) failure of any number of processes and communication paths. The Paxon algorithm is suitable for systems with modest reliability requirements that do not justify the expense of an extremely fault-tolerant, real-time implementation.

If the state machine is executed with an algorithm that guarantees bounded-time response, then time can be made part of the state, and machine actions can be triggered by the passage of time. For example, consider a system for granting ownership of resources. The state can include the time at which a client was granted a resource, and the state machine can automatically execute a command to revoke ownership if the client has held the resource too long.

With the Paxon algorithm, time cannot be made part of the state in such a natural way. If failures occur, it can take arbitrarily long to execute a command (pass a decree), and one command can be executed before (appear earlier in the sequence of decrees than) another command that was issued earlier. However, a state machine can still use real time the same way the Paxon Parliament did. For example, the method described in Section 3.3.3 for deciding who was the current cheese inspector can be used to decide who is the current owner of a resource.

4.2 Commit Protocols

The Paxon Synod protocol is similar to standard three-phase commit protocols [Bernstein et al. 1987; Skeen 1982]. A Paxon ballot and a three-phase commit protocol both involve the exchange of five messages between a coordinator (the president) and the other quorum members (legislators). A commit protocol chooses one of two values—*commit* or *abort*—while the Synod protocol chooses an arbitrary decree. To convert a commit protocol to a Synod protocol, one sends the decree in the initial round of messages. A *commit* decision means that this decree was passed, and an *abort* decision means that the "olive-day" decree was passed.

The Synod protocol differs from a converted commit protocol because the decree is not sent until the second phase. This allows the corresponding parliamentary protocol to execute the first phase just once for all decrees, so the exchange of only three messages is needed to pass each individual decree.

The theorems on which the Synod protocol is based are similar to results obtained

[14] These algorithms were derived from the military protocols of another Mediterranean state.

by Dwork, Lynch, and Stockmeyer [Dwork et al. 1988]. However, their algorithms execute ballots sequentially in separate rounds, and they seem to be unrelated to the Synod protocol.

Much research has been done in the field since this article was written. The state-machine approach has been surveyed by Schneider [1990]. The recovery protocol by Keidar and Dolev [1996] and the totally-ordered broadcast algorithm of Fekete et al. [1997] are quite similar to the Paxon protocol described here. The author was also apparently unaware that the view management protocol by Oki and Liskov [1988] seems to be equivalent to the Paxon protocol.

Many of the refinements presented in this submission have also appeared in contemporary or subsequent articles. The method of delegation described in Section 3.3.3 is very similar to the leases mechanism of Gray and Cheriton [1989]. The technique of Section 3.3.4 in which the Paxons satisfy the monotonicity condition by using decree numbers is described by Ladin et al. [1992]. The technique of Section 3.3.6 for adding new legislators was also given by Schneider [1990].

K. M.

Appendix: Proof of Consistency of the Synodic Protocol

A1 The Basic Protocol

The Synod's basic protocol, described informally in Section 2.3, is stated here using modern algorithmic notation. We begin with the variables that a priest p must maintain. First come the variables that represent information kept in his ledger. (For convenience, the vote $prevVote[p]$ used in Section 2.3 is replaced by its components $prevBal[p]$ and $prevDec[p]$.)

- $outcome[p]$ — The decree written in p's ledger, or BLANK if there is nothing written there yet.
- $lastTried[p]$ — The number of the last ballot that p tried to begin, or $-\infty$ if there was none.
- $prevBal[p]$ — The number of the last ballot in which p voted, or $-\infty$ if he never voted.
- $prevDec[p]$ — The decree for which p last voted, or BLANK if p never voted.
- $nextBal[p]$ — The number of the last ballot in which p agreed to participate, or $-\infty$ if he has never agreed to participate in a ballot.

Next come variables representing information that priest p could keep on a slip of paper:

- $status[p]$ — One of the following values:
  - *idle* — Not conducting or trying to begin a ballot
  - *trying* — Trying to begin ballot number $lastTried[p]$
  - *polling* — Now conducting ballot number $lastTried[p]$

  If p has lost his slip of paper, then $status[p]$ is assumed to equal *idle* and the values of the following four variables are irrelevant.
- $prevVotes[p]$ — The set of votes received in *LastVote* messages for the current ballot (the one with ballot number $lastTried[p]$).

quorum[*p*]	If *status*[*p*] = *polling*, then the set of priests forming the quorum of the current ballot; otherwise, meaningless.
voters[*p*]	If *status*[*p*] = *polling*, then the set of quorum members from whom *p* has received *Voted* messages in the current ballot; otherwise, meaningless.
decree[*p*]	If *status*[*p*] = *polling*, then the decree of the current ballot; otherwise, meaningless.

There is also the history variable $\mathcal{B}$, which is the set of ballots that have been started and their progress—namely, which priests have cast votes. (A history variable is one used in the development and proof of an algorithm, but not actually implemented.)

Next come the actions that priest *p* may take. These actions are assumed to be atomic, meaning that once an action is begun, it must be completed before priest *p* begins any other action. An action is described by an enabling condition and a list of effects. The enabling condition describes when the action can be performed; actions that receive a message are enabled whenever a messenger has arrived with the appropriate message. The list of effects describes how the action changes the algorithm's variables and what message, if any, it sends. (Each individual action sends at most one message.)

Recall that ballot numbers were partitioned among the priests. For any ballot number *b*, the Paxons defined *owner*(*b*) to be the priest who was allowed to use that ballot number.

The actions in the basic protocol are allowed actions; the protocol does not require that a priest ever do anything. No attempt at efficiency has been made; the actions allow *p* to do silly things, such as sending another *BeginBallot* message to a priest from whom he has already received a *LastVote* message.

Try New Ballot

Always enabled.

- Set *lastTried*[*p*] to any ballot number *b*, greater than its previous value, such that *owner*(*b*) = *p*.
- Set *status*[*p*] to *trying*.
- Set *prevVotes*[*p*] to $\emptyset$.

Send *NextBallot* **Message**

Enabled whenever *status*[*p*]=*trying*.

- Send a *NextBallot*(*lastTried*[*p*]) message to any priest.

Receive *NextBallot*(*b*) **Message**

If $b \geq$ *nextBal*[*p*] then

- Set *nextBal*[*p*] to *b*.

Send *LastVote* **Message**

Enabled whenever *nextBal*[*p*] > *prevBal*[*p*].

- Send a *LastVote*(*nextBal*[*p*], *v*) message to priest *owner*(*nextBal*[*p*]), where $v_{pst} = p$, $v_{bal} =$ *prevBal*[*p*], and $v_{dec} =$ *prevDec*[*p*].

Receive *LastVote*(b, v) **Message**

If $b = \mathit{lastTried}[p]$ and $\mathit{status}[p] = \mathit{trying}$, then

- Set $\mathit{prevVotes}[p]$ to the union of its original value and $\{v\}$.

Start Polling Majority Set Q

Enabled when $\mathit{status}[p] = \mathit{trying}$ and $Q \subseteq \{v_{pst} : v \in \mathit{prevVotes}[p]\}$, where Q is a majority set.

- Set $\mathit{status}[p]$ to *polling*.
- Set $\mathit{quorum}[p]$ to Q.
- Set $\mathit{voters}[p]$ to $\emptyset$.
- Set $\mathit{decree}[p]$ to a decree d chosen as follows: Let v be the maximum element of $\mathit{prevVotes}[p]$. If $v_{bal} \neq -\infty$ then $d = v_{dec}$, else d can equal any decree.
- Set $\mathcal{B}$ to the union of its former value and $\{B\}$, where $B_{dec} = d$, $B_{qrm} = Q$, $B_{vot} = \emptyset$, and $B_{bal} = \mathit{lastTried}[p]$.

Send *BeginBallot* **Message**

Enabled when $\mathit{status}[p] = \mathit{polling}$.

- Send a *BeginBallot*$(\mathit{lastTried}[p],\ \mathit{decree}[p])$ message to any priest in $\mathit{quorum}[p]$.

Receive *BeginBallot*$(b,\ d)$ **Message**

If $b = \mathit{nextBal}[p] > \mathit{prevBal}[p]$ then

- Set $\mathit{prevBal}[p]$ to b.
- Set $\mathit{prevDec}[p]$ to d.
- If there is a ballot B in $\mathcal{B}$ with $B_{bal} = b$ [there will be], then choose any such B [there will be only one] and let the new value of $\mathcal{B}$ be obtained from its old value by setting B_{vot} equal to the union of its old value and $\{p\}$.

Send *Voted* **Message**

Enabled whenever $\mathit{prevBal}[p] \neq -\infty$.

- Send a *Voted*$(\mathit{prevBal}[p],\ p)$ message to $\mathit{owner}(\mathit{prevBal}[p])$.

Receive *Voted*$(b,\ q)$ **Message**

If $b = \mathit{lastTried}[p]$ and $\mathit{status}[p] = \mathit{polling}$, then

- Set $\mathit{voters}[p]$ to the union of its old value and $\{q\}$

Succeed

Enabled whenever $\mathit{status}[p] = \mathit{polling}$, $\mathit{quorum}[p] \subseteq \mathit{voters}[p]$, and $\mathit{outcome}[p] =$ BLANK.

- Set $\mathit{outcome}[p]$ to $\mathit{decree}[p]$.

Send *Success* **Message**

Enabled whenever $\mathit{outcome}[p] \neq$ BLANK.

- Send a *Success*$(\mathit{outcome}[p])$ message to any priest.

Receive *Success*(d) **Message**

If $\mathit{outcome}[p] =$ BLANK, then

– Set $outcome[p]$ to d.

This algorithm is an abstract description of the real protocol performed by Paxon priests. Do the algorithm's actions accurately model the actions of the real priests? There were three kinds of actions that a priest could perform "atomically": receiving a message, writing a note or ledger entry, and sending a message. Each of these is represented by a single action of the algorithm, except that **Receive** actions both receive a message and set a variable. We can pretend that the receipt of a message occurred when a priest acted upon the message; if he left the Chamber before acting upon it, then we can pretend that the message was never received. Since this pretense does not affect the consistency condition, we can infer the consistency of the basic Synod protocol from the consistency of the algorithm.

A2 Proof of Consistency

To prove the consistency condition, it is necessary to show that whenever $outcome[p]$ and $outcome[q]$ are both different from BLANK, they are equal. A rigorous correctness proof requires a complete description of the algorithm. The description given above is almost complete. Missing is a variable $\mathcal{M}$ whose value is the multiset of all messages in transit.[15] Each **Send** action adds a message to this multiset and each **Receive** action removes one. Also needed are actions to represent the loss and duplication of messages, as well as a **Forget** action that represents a priest losing his slip of paper.

With these additions, we get an algorithm that defines a set of possible behaviors, in which each change of state corresponds to one of the allowed actions. The Paxons proved correctness by finding a predicate I such that

(1) I is true initially.

(2) I implies the desired correctness condition.

(3) Each allowed action leaves I true.

The predicate I was written as a conjunction $I1 \wedge \ldots \wedge I7$, where $I1$–$I5$ were in turn the conjunction of predicates $I1(p)$–$I5(p)$ for all priests p. Although most variables are mentioned in several of the conjuncts, each variable except $status[p]$ is naturally associated with one conjunct, and each conjunct can be thought of as a constraint on its associated variables. The definitions of the individual conjuncts of I are given below, where a list of items marked by $\wedge$ symbols denotes the conjunction of those items. The variables associated with a conjunct are listed in bracketed comments.

$I1(p) \triangleq$ [Associated variable: $outcome[p]$]

$(outcome[p] \neq \text{BLANK}) \Rightarrow \exists B \in \mathcal{B} : (B_{qrm} \subseteq B_{vot}) \wedge (B_{dec} = outcome[p])$

$I2(p) \triangleq$ [Associated variable: $lastTried[p]$]

$\wedge\ owner(lastTried[p]) = p$
$\wedge\ \forall B \in \mathcal{B} : (owner(B_{bal}) = p) \Rightarrow$
$\qquad \wedge\ B_{bal} \leq lastTried[p]$
$\qquad \wedge\ (status[p] = trying) \Rightarrow (B_{bal} < lastTried[p])$

[15] A multiset is a set that may contain multiple copies of the same element.

$I3(p) \triangleq$ [Associated variables: $prevBal[p]$, $prevDec[p]$, $nextBal[p]$]

$\land\ prevBal[p] = MaxVote(\infty, p, \mathcal{B})_{bal}$
$\land\ prevDec[p] = MaxVote(\infty, p, \mathcal{B})_{dec}$
$\land\ nextBal[p] \geq prevBal[p]$

$I4(p) \triangleq$ [Associated variable: $prevVotes[p]$]

$(status[p] \neq idle) \Rightarrow$
$\quad \forall v \in prevVotes[p] : \land\ v = MaxVote(lastTried[p], v_{pst}, \mathcal{B})$
$\qquad\qquad\qquad\qquad \land\ nextBal[v_{pst}] \geq lastTried[p]$

$I5(p) \triangleq$ [Associated variables: $quorum[p]$, $voters[p]$, $decree[p]$]

$(status[p] = polling) \Rightarrow$
$\quad \land\ quorum[p] \subseteq \{v_{pst} : v \in prevVotes[p]\}$
$\quad \land\ \exists B \in \mathcal{B} : \land\ quorum[p] = B_{qrm}$
$\qquad\qquad\qquad \land\ decree[p] = B_{dec}$
$\qquad\qquad\qquad \land\ voters[p] \subseteq B_{vot}$
$\qquad\qquad\qquad \land\ lastTried[p] = B_{bal}$

$I6 \triangleq$ [Associated variable: $\mathcal{B}$]

$\land\ B1(\mathcal{B}) \land B2(\mathcal{B}) \land B3(\mathcal{B})$
$\land\ \forall B \in \mathcal{B} :\ B_{qrm}$ is a majority set

$I7 \triangleq$ [Associated variable: $\mathcal{M}$]

$\land\ \forall NextBallot(b) \in \mathcal{M} : (b \leq lastTried[owner(b)])$
$\land\ \forall LastVote(b, v) \in \mathcal{M} : \land\ v = MaxVote(b, v_{pst}, \mathcal{B})$
$\qquad\qquad\qquad\qquad\qquad\qquad \land\ nextBal[v_{pst}] \geq b$
$\land\ \forall BeginBallot(b, d) \in \mathcal{M} : \exists B \in \mathcal{B} : (B_{bal} = b) \land (B_{dec} = d)$
$\land\ \forall Voted(b, p) \in \mathcal{M} : \exists B \in \mathcal{B} : (B_{bal} = b) \land (p \in B_{vot})$
$\land\ \forall Success(d) \in \mathcal{M} : \exists p : outcome[p] = d \neq \text{BLANK}$

The Paxons had to prove that I satisfies the three conditions given above. The first condition, that I holds initially, requires checking that each conjunct is true for the initial values of all the variables. While not stated explicitly, these initial values can be inferred from the variables' descriptions, and checking the first condition is straightforward. The second condition, that I implies consistency, follows from $I1$, the first conjunct of $I6$, and Theorem 1. The hard part was proving the third condition, the invariance of I, which meant proving that I is left true by every action. This condition is proved by showing that, for each conjunct of I, executing any action when I is true leaves that conjunct true. The proofs are sketched below.

$I1(p)$ $\mathcal{B}$ is changed only by adding a new ballot or adding a new priest to B_{vot} for some $B \in \mathcal{B}$, neither of which can falsify $I1(p)$. The value of $outcome[p]$ is changed only by the **Succeed** and **Receive** *Success* **Message** actions. The enabling condition and $I5(p)$ imply that $I1(p)$ is left true by the **Succeed** action. The enabling condition, $I1(p)$, and the last conjunct of $I7$ imply that $I1(p)$ is left true by the **Receive** *Success* **Message** action.

$I2(p)$ This conjunct depends only on $lastTried[p]$, $status[p]$, and $\mathcal{B}$. Only the **Try New Ballot** action changes $lastTried[p]$, and only that action can set $status[p]$ to *trying*. Since the action increases $lastTried[p]$ to a value b with $owner(b) = p$, it leaves $I2(p)$ true. A completely new element is added to $\mathcal{B}$ only by a **Start Polling** action; the first conjunct of $I2(p)$ and the specification of the action imply that adding this new element does not falsify the second conjunct of $I2(p)$. The only other way $\mathcal{B}$ is changed is by adding a new priest to B_{vot} for some $B \in \mathcal{B}$, which does not affect $I2(p)$.

$I3(p)$ Since votes are never removed from $\mathcal{B}$, the only action that can change $MaxVote(\infty, p, \mathcal{B})$ is one that adds to $\mathcal{B}$ a vote cast by p. Only a **Receive** *BeginBallot* **Message** action can do that, and only that action changes $prevBal[p]$ and $prevDec[p]$. The *BeginBallot* conjunct of $I7$ implies that this action actually does add a vote to $\mathcal{B}$, and $B1(\mathcal{B})$ (the first conjunct of $I6$) implies that there is only one ballot to which the vote can be added. The enabling condition, the assumption that $I3(p)$ holds before executing the action, and the definition of *MaxVote* then imply that the action leaves the first two conjuncts of $I3(p)$ true. The third conjunct is left true because $prevBal[p]$ is changed only by setting it to $nextBal[p]$, and $nextBal[p]$ is never decreased.

$I4(p)$ This conjunct depends only upon the values of $status[p]$, $prevVotes[p]$, $lastTried[p]$, $nextBal[q]$ for some priests q, and $\mathcal{B}$. The value of $status[p]$ is changed from *idle* to not *idle* only by a **Try New Ballot** action, which sets $prevVotes[p]$ to $\emptyset$, making $I4(p)$ vacuously true. The only other actions that change $prevVotes[p]$ are the **Forget** action, which leaves $I4(p)$ true because it sets $status[p]$ to *idle*, and the **Receive** *LastVote* **Message** action. It follows from the enabling condition and the *LastVote* conjunct of $I7$ that the **Receive** *LastVote* **Message** action preserves $I4(p)$. The value of $lastTried[p]$ is changed only by the **Try New Ballot** action, which leaves $I4(p)$ true because it sets $status[p]$ to *trying*. The value of $nextBal[q]$ can only increase, which cannot make $I4(p)$ false. Finally, $MaxVote(lastTried[p], v_{pst}, \mathcal{B})$ can be changed only if v_{pst} is added to B_{vot} for some $B \in \mathcal{B}$ with $B_{bal} < lastTried[p]$. But v_{pst} is added to B_{vot} (by a **Receive** *BeginBallot* **Message** action) only if $nextBal[v_{pst}] = B_{bal}$, in which case $I4(p)$ implies that $B_{bal} \geq lastTried[p]$.

$I5(p)$ The value of $status[p]$ is set to *polling* only by the **Start Polling** action. This action's enabling condition guarantees that the first conjunct becomes true, and it adds the ballot to $\mathcal{B}$ that makes the second conjunct true. No other action changes $quorum[p]$, $decree[p]$, or $lastTried[p]$ while leaving $status[p]$ equal to *polling*. The value of $prevVotes[p]$ cannot be changed while $status[p] = polling$, and $\mathcal{B}$ is changed only by adding new elements or by adding a new priest to B_{vot}. The only remaining possibility for falsifying $I5(p)$ is the addition of a new element to $voters[p]$ by the **Receive** *Voted* **Message** action. The *Voted* conjunct of $I7$, $B1(\mathcal{B})$ (the first conjunct of $I6$), and the action's enabling condition imply that the element added to $voters[p]$ is in B_{vot}, where B is the ballot whose existence is asserted in $I5(p)$.

$I6$ Since B_{bal} and B_{qrm} are never changed for any $B \in \mathcal{B}$, the only way $B1(\mathcal{B})$, $B2(\mathcal{B})$, and the second conjunct of $I6$ can be falsified is by adding a new ballot to

$\mathcal{B}$, which is done only by the **Start Polling Majority Set** Q action when *status*[p] equals *trying*. It follows from the second conjunct of $I2(p)$ that this action leaves $B1(\mathcal{B})$ true; and the assertion, in the enabling condition, that Q is a majority set implies that the action leaves $B2(\mathcal{B})$ and the second conjunct of $I6$ true. There are two possible ways of falsifying $B3(\mathcal{B})$: changing $MaxVote(B_{bal}, B_{qrm}, \mathcal{B})$ by adding a new vote to $\mathcal{B}$, and adding a new ballot to $\mathcal{B}$. A new vote is added only by the **Receive** *BeginBallot* **Message** action, and $I3(p)$ implies that the action adds a vote later than any other vote cast by p in $\mathcal{B}$, so it cannot change $MaxVote(B_{bal}, B_{qrm}, \mathcal{B})$ for any B in $\mathcal{B}$. Conjunct $I4(p)$ implies that the new ballot added by the **Start Polling** action does not falsify $B3(\mathcal{B})$.

I7 $I7$ can be falsified either by adding a new message to $\mathcal{M}$ or by changing the value of another variable on which $I7$ depends. Since *lastTried*[p] and *nextBal*[p] are never decreased, changing them cannot make $I7$ false. Since *outcome*[p] is never changed if its value is not BLANK, changing it cannot falsify $I7$. Since $\mathcal{B}$ is changed only by adding ballots and adding votes, the only change to it that can make $I7$ false is the addition of a vote by v_{pst} that makes the $LastVote(b, v)$ conjunct false by changing $MaxVote(b, v_{pst}, \mathcal{B})$. This can happen only if v_{pst} votes in a ballot B with $B_{bal} < b$. But v_{pst} can vote only in ballot number $nextBal[v_{pst}]$, and the assumption that this conjunct holds initially implies that $nextBal[v_{pst}] \geq b$. Therefore, we need check only that every message that is sent satisfies the condition in the appropriate conjunct of $I7$.

NextBallot: Follows from the definition of the **Send** *NextBallot* **Message** action and the first conjunct of $I2(p)$.

LastVote: The enabling condition of the **Send** *LastVote* **Message** action and $I3(p)$ imply that $MaxVote(nextBal[p], p, \mathcal{B}) = MaxVote(\infty, p, \mathcal{B})$, from which it follows that the *LastVote* message sent by the action satisfies the condition in $I7$.

BeginBallot: Follows from $I5(p)$ and the definition of the **Send** *BeginBallot* **Message** action.

Voted: Follows from $I3(p)$, the definition of *MaxVote*, and the definition of the **Send** *Voted* **Message** action.

Success: Follows from the definition of **Send** *Success* **Message**.

ACKNOWLEDGMENTS

Daniel Duchamp pointed out to me the need for a new state-machine implementation. Discussions with Martín Abadi, Andy Hisgen, Tim Mann, and Garret Swart led me to Paxos. *Λεωνίδας Γκίμπας* provided invaluable assistance with the Paxon dialect.

References

Bernstein, P. A., Hadzilacos, V., and Goodman, N. 1987. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, Reading, Massachusetts.

De Prisco, R., Lampson, B., and Lynch, N. 1997. Revisiting the Paxos algorithm. In M. Mavronicolas and P. Tsigas (Eds.), *Proceedings of the 11th International Workshop on Distributed Algorithms (WDAG 97)*, Volume 1320 of *Lecture Notes in Computer*

Science, Saarbruken, Germany, pp. 111–125. Springer-Verlag.

Dijkstra, E. W. 1974. Self-stabilizing systems in spite of distributed control. *Commun. ACM 17,* 11 (Nov.), 643–644.

Dwork, C., Lynch, N., and Stockmeyer, L. 1988. Consensus in the presence of partial synchrony. *Journal of the ACM 35,* 2 (April), 288–323.

Fekete, A., Lynch, N., and Shvartsman, A. 1997. Specifying and using a partitionable group communication service. In *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Distributed Computing*, pp. 53–62. ACM Press.

Fischer, M. J., Lynch, N., and Paterson, M. S. 1985. Impossibility of distributed consensus with one faulty process. *Journal of the ACM 32,* 2 (April), 374–382.

Gray, C. G. and Cheriton, D. R. 1989. Leases: An efficient fault-toerant mechanism for distributed file cache consistency. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, New York, pp. 202–210. ACM.

Keidar, I. and Dolev, D. 1996. Efficient message ordering in dynamic networks. In *Proceedings of the* 15th *Annual ACM Symposium on Principles of Distributed Computing*. ACM.

Ladin, R., Liskov, B., Shrira, L., and Ghemawat, S. 1992. Providing high availability using lazy replication. *ACM Transactions on Computer Systems 10,* 4 (Nov.), 360–391.

Lamport, L. 1978. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM 21,* 7 (July), 558–565.

Lamport, L. 1984. Using time instead of timeout for fault-tolerant distributed systems. *ACM Trans. on Programm. Lang. Syst. 6,* 2 (April), 254–280.

Lampson, B. W. 1996. How to build a highly available system using consensus. In O. Babaoglu and K. Marzullo (Eds.), *Distributed Algorithms*, Volume 1151 of *Lecture Notes in Computer Science*, Berlin, pp. 1–17. Springer-Verlag.

Oki, B. M. and Liskov, B. H. 1988. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, pp. 8–17. ACM Press.

Schneider, F. B. 1990. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys 22,* 4 (Dec.), 299–319.

Skeen, M. D. 1982. Crash recovery in a distributed database system. Ph. D. thesis, University of California, Berkeley.

Received January 1990; Accepted March 1998

Chapter 6

Paxos Made Simple

Authors:
Leslie Lamport

Publication year: 2001

Paxos Made Simple

Leslie Lamport

01 Nov 2001

Abstract

The Paxos algorithm, when presented in plain English, is very simple.

Contents

1 Introduction **1**

2 The Consensus Algorithm **1**

2.1 The Problem 1

2.2 Choosing a Value 2

2.3 Learning a Chosen Value 6

2.4 Progress 7

2.5 The Implementation 7

3 Implementing a State Machine **8**

References **11**

1 Introduction

The Paxos algorithm for implementing a fault-tolerant distributed system has been regarded as difficult to understand, perhaps because the original presentation was Greek to many readers [5]. In fact, it is among the simplest and most obvious of distributed algorithms. At its heart is a consensus algorithm—the "synod" algorithm of [5]. The next section shows that this consensus algorithm follows almost unavoidably from the properties we want it to satisfy. The last section explains the complete Paxos algorithm, which is obtained by the straightforward application of consensus to the state machine approach for building a distributed system—an approach that should be well-known, since it is the subject of what is probably the most often-cited article on the theory of distributed systems [4].

2 The Consensus Algorithm

2.1 The Problem

Assume a collection of processes that can propose values. A consensus algorithm ensures that a single one among the proposed values is chosen. If no value is proposed, then no value should be chosen. If a value has been chosen, then processes should be able to learn the chosen value. The safety requirements for consensus are:

- Only a value that has been proposed may be chosen,
- Only a single value is chosen, and
- A process never learns that a value has been chosen unless it actually has been.

We won't try to specify precise liveness requirements. However, the goal is to ensure that some proposed value is eventually chosen and, if a value has been chosen, then a process can eventually learn the value.

We let the three roles in the consensus algorithm be performed by three classes of agents: *proposers*, *acceptors*, and *learners*. In an implementation, a single process may act as more than one agent, but the mapping from agents to processes does not concern us here.

Assume that agents can communicate with one another by sending messages. We use the customary asynchronous, non-Byzantine model, in which:

- Agents operate at arbitrary speed, may fail by stopping, and may restart. Since all agents may fail after a value is chosen and then restart, a solution is impossible unless some information can be remembered by an agent that has failed and restarted.

- Messages can take arbitrarily long to be delivered, can be duplicated, and can be lost, but they are not corrupted.

2.2 Choosing a Value

The easiest way to choose a value is to have a single acceptor agent. A proposer sends a proposal to the acceptor, who chooses the first proposed value that it receives. Although simple, this solution is unsatisfactory because the failure of the acceptor makes any further progress impossible.

So, let's try another way of choosing a value. Instead of a single acceptor, let's use multiple acceptor agents. A proposer sends a proposed value to a set of acceptors. An acceptor may *accept* the proposed value. The value is chosen when a large enough set of acceptors have accepted it. How large is large enough? To ensure that only a single value is chosen, we can let a large enough set consist of any majority of the agents. Because any two majorities have at least one acceptor in common, this works if an acceptor can accept at most one value. (There is an obvious generalization of a majority that has been observed in numerous papers, apparently starting with [3].)

In the absence of failure or message loss, we want a value to be chosen even if only one value is proposed by a single proposer. This suggests the requirement:

P1. An acceptor must accept the first proposal that it receives.

But this requirement raises a problem. Several values could be proposed by different proposers at about the same time, leading to a situation in which every acceptor has accepted a value, but no single value is accepted by a majority of them. Even with just two proposed values, if each is accepted by about half the acceptors, failure of a single acceptor could make it impossible to learn which of the values was chosen.

P1 and the requirement that a value is chosen only when it is accepted by a majority of acceptors imply that an acceptor must be allowed to accept more than one proposal. We keep track of the different proposals that an acceptor may accept by assigning a (natural) number to each proposal, so a proposal consists of a proposal number and a value. To prevent confusion, we require that different proposals have different numbers. How this is

achieved depends on the implementation, so for now we just assume it. A value is chosen when a single proposal with that value has been accepted by a majority of the acceptors. In that case, we say that the proposal (as well as its value) has been chosen.

We can allow multiple proposals to be chosen, but we must guarantee that all chosen proposals have the same value. By induction on the proposal number, it suffices to guarantee:

P2. If a proposal with value v is chosen, then every higher-numbered proposal that is chosen has value v.

Since numbers are totally ordered, condition P2 guarantees the crucial safety property that only a single value is chosen.

To be chosen, a proposal must be accepted by at least one acceptor. So, we can satisfy P2 by satisfying:

P2^a. If a proposal with value v is chosen, then every higher-numbered proposal accepted by any acceptor has value v.

We still maintain P1 to ensure that some proposal is chosen. Because communication is asynchronous, a proposal could be chosen with some particular acceptor c never having received any proposal. Suppose a new proposer "wakes up" and issues a higher-numbered proposal with a different value. P1 requires c to accept this proposal, violating P2^a. Maintaining both P1 and P2^a requires strengthening P2^a to:

P2^b. If a proposal with value v is chosen, then every higher-numbered proposal issued by any proposer has value v.

Since a proposal must be issued by a proposer before it can be accepted by an acceptor, P2^b implies P2^a, which in turn implies $P2$.

To discover how to satisfy P2^b, let's consider how we would prove that it holds. We would assume that some proposal with number m and value v is chosen and show that any proposal issued with number $n > m$ also has value v. We would make the proof easier by using induction on n, so we can prove that proposal number n has value v under the additional assumption that every proposal issued with a number in $m \mathinner{..} (n-1)$ has value v, where $i \mathinner{..} j$ denotes the set of numbers from i through j. For the proposal numbered m to be chosen, there must be some set C consisting of a majority of acceptors such that every acceptor in C accepted it. Combining this with the induction assumption, the hypothesis that m is chosen implies:

Every acceptor in C has accepted a proposal with number in $m\,..\,(n-1)$, and every proposal with number in $m\,..\,(n-1)$ accepted by any acceptor has value v.

Since any set S consisting of a majority of acceptors contains at least one member of C, we can conclude that a proposal numbered n has value v by ensuring that the following invariant is maintained:

P2^c. For any v and n, if a proposal with value v and number n is issued, then there is a set S consisting of a majority of acceptors such that either (a) no acceptor in S has accepted any proposal numbered less than n, or (b) v is the value of the highest-numbered proposal among all proposals numbered less than n accepted by the acceptors in S.

We can therefore satisfy P2^b by maintaining the invariance of P2^c.

To maintain the invariance of P2^c, a proposer that wants to issue a proposal numbered n must learn the highest-numbered proposal with number less than n, if any, that has been or will be accepted by each acceptor in some majority of acceptors. Learning about proposals already accepted is easy enough; predicting future acceptances is hard. Instead of trying to predict the future, the proposer controls it by extracting a promise that there won't be any such acceptances. In other words, the proposer requests that the acceptors not accept any more proposals numbered less than n. This leads to the following algorithm for issuing proposals.

1. A proposer chooses a new proposal number n and sends a request to each member of some set of acceptors, asking it to respond with:

   (a) A promise never again to accept a proposal numbered less than n, and

   (b) The proposal with the highest number less than n that it has accepted, if any.

   I will call such a request a *prepare* request with number n.

2. If the proposer receives the requested responses from a majority of the acceptors, then it can issue a proposal with number n and value v, where v is the value of the highest-numbered proposal among the responses, or is any value selected by the proposer if the responders reported no proposals.

A proposer issues a proposal by sending, to some set of acceptors, a request that the proposal be accepted. (This need not be the same set of acceptors that responded to the initial requests.) Let's call this an *accept* request.

This describes a proposer's algorithm. What about an acceptor? It can receive two kinds of requests from proposers: *prepare* requests and *accept* requests. An acceptor can ignore any request without compromising safety. So, we need to say only when it is allowed to respond to a request. It can always respond to a *prepare* request. It can respond to an *accept* request, accepting the proposal, iff it has not promised not to. In other words:

P1^a. An acceptor can accept a proposal numbered n iff it has not responded to a *prepare* request having a number greater than n.

Observe that P1^a subsumes P1.

We now have a complete algorithm for choosing a value that satisfies the required safety properties—assuming unique proposal numbers. The final algorithm is obtained by making one small optimization.

Suppose an acceptor receives a *prepare* request numbered n, but it has already responded to a *prepare* request numbered greater than n, thereby promising not to accept any new proposal numbered n. There is then no reason for the acceptor to respond to the new *prepare* request, since it will not accept the proposal numbered n that the proposer wants to issue. So we have the acceptor ignore such a *prepare* request. We also have it ignore a *prepare* request for a proposal it has already accepted.

With this optimization, an acceptor needs to remember only the highest-numbered proposal that it has ever accepted and the number of the highest-numbered *prepare* request to which it has responded. Because P2^c must be kept invariant regardless of failures, an acceptor must remember this information even if it fails and then restarts. Note that the proposer can always abandon a proposal and forget all about it—as long as it never tries to issue another proposal with the same number.

Putting the actions of the proposer and acceptor together, we see that the algorithm operates in the following two phases.

Phase 1. (a) A proposer selects a proposal number n and sends a *prepare* request with number n to a majority of acceptors.

(b) If an acceptor receives a *prepare* request with number n greater than that of any *prepare* request to which it has already responded, then it responds to the request with a promise not to accept any more proposals numbered less than n and with the highest-numbered proposal (if any) that it has accepted.

Phase 2. (a) If the proposer receives a response to its *prepare* requests (numbered n) from a majority of acceptors, then it sends an *accept* request to each of those acceptors for a proposal numbered n with a value v, where v is the value of the highest-numbered proposal among the responses, or is any value if the responses reported no proposals.

(b) If an acceptor receives an *accept* request for a proposal numbered n, it accepts the proposal unless it has already responded to a *prepare* request having a number greater than n.

A proposer can make multiple proposals, so long as it follows the algorithm for each one. It can abandon a proposal in the middle of the protocol at any time. (Correctness is maintained, even though requests and/or responses for the proposal may arrive at their destinations long after the proposal was abandoned.) It is probably a good idea to abandon a proposal if some proposer has begun trying to issue a higher-numbered one. Therefore, if an acceptor ignores a *prepare* or *accept* request because it has already received a *prepare* request with a higher number, then it should probably inform the proposer, who should then abandon its proposal. This is a performance optimization that does not affect correctness.

2.3 Learning a Chosen Value

To learn that a value has been chosen, a learner must find out that a proposal has been accepted by a majority of acceptors. The obvious algorithm is to have each acceptor, whenever it accepts a proposal, respond to all learners, sending them the proposal. This allows learners to find out about a chosen value as soon as possible, but it requires each acceptor to respond to each learner—a number of responses equal to the product of the number of acceptors and the number of learners.

The assumption of non-Byzantine failures makes it easy for one learner to find out from another learner that a value has been accepted. We can have the acceptors respond with their acceptances to a distinguished learner, which in turn informs the other learners when a value has been chosen. This approach requires an extra round for all the learners to discover the chosen value. It is also less reliable, since the distinguished learner could fail. But it requires a number of responses equal only to the sum of the number of acceptors and the number of learners.

More generally, the acceptors could respond with their acceptances to some set of distinguished learners, each of which can then inform all the learners when a value has been chosen. Using a larger set of distinguished

learners provides greater reliability at the cost of greater communication complexity.

Because of message loss, a value could be chosen with no learner ever finding out. The learner could ask the acceptors what proposals they have accepted, but failure of an acceptor could make it impossible to know whether or not a majority had accepted a particular proposal. In that case, learners will find out what value is chosen only when a new proposal is chosen. If a learner needs to know whether a value has been chosen, it can have a proposer issue a proposal, using the algorithm described above.

2.4 Progress

It's easy to construct a scenario in which two proposers each keep issuing a sequence of proposals with increasing numbers, none of which are ever chosen. Proposer p completes phase 1 for a proposal number n_1. Another proposer q then completes phase 1 for a proposal number $n_2 > n_1$. Proposer p's phase 2 *accept* requests for a proposal numbered n_1 are ignored because the acceptors have all promised not to accept any new proposal numbered less than n_2. So, proposer p then begins and completes phase 1 for a new proposal number $n_3 > n_2$, causing the second phase 2 *accept* requests of proposer q to be ignored. And so on.

To guarantee progress, a distinguished proposer must be selected as the only one to try issuing proposals. If the distinguished proposer can communicate successfully with a majority of acceptors, and if it uses a proposal with number greater than any already used, then it will succeed in issuing a proposal that is accepted. By abandoning a proposal and trying again if it learns about some request with a higher proposal number, the distinguished proposer will eventually choose a high enough proposal number.

If enough of the system (proposer, acceptors, and communication network) is working properly, liveness can therefore be achieved by electing a single distinguished proposer. The famous result of Fischer, Lynch, and Patterson [1] implies that a reliable algorithm for electing a proposer must use either randomness or real time—for example, by using timeouts. However, safety is ensured regardless of the success or failure of the election.

2.5 The Implementation

The Paxos algorithm [5] assumes a network of processes. In its consensus algorithm, each process plays the role of proposer, acceptor, and learner. The algorithm chooses a leader, which plays the roles of the distinguished

proposer and the distinguished learner. The Paxos consensus algorithm is precisely the one described above, where requests and responses are sent as ordinary messages. (Response messages are tagged with the corresponding proposal number to prevent confusion.) Stable storage, preserved during failures, is used to maintain the information that the acceptor must remember. An acceptor records its intended response in stable storage before actually sending the response.

All that remains is to describe the mechanism for guaranteeing that no two proposals are ever issued with the same number. Different proposers choose their numbers from disjoint sets of numbers, so two different proposers never issue a proposal with the same number. Each proposer remembers (in stable storage) the highest-numbered proposal it has tried to issue, and begins phase 1 with a higher proposal number than any it has already used.

3 Implementing a State Machine

A simple way to implement a distributed system is as a collection of clients that issue commands to a central server. The server can be described as a deterministic state machine that performs client commands in some sequence. The state machine has a current state; it performs a step by taking as input a command and producing an output and a new state. For example, the clients of a distributed banking system might be tellers, and the state-machine state might consist of the account balances of all users. A withdrawal would be performed by executing a state machine command that decreases an account's balance if and only if the balance is greater than the amount withdrawn, producing as output the old and new balances.

An implementation that uses a single central server fails if that server fails. We therefore instead use a collection of servers, each one independently implementing the state machine. Because the state machine is deterministic, all the servers will produce the same sequences of states and outputs if they all execute the same sequence of commands. A client issuing a command can then use the output generated for it by any server.

To guarantee that all servers execute the same sequence of state machine commands, we implement a sequence of separate instances of the Paxos consensus algorithm, the value chosen by the ith instance being the ith state machine command in the sequence. Each server plays all the roles (proposer, acceptor, and learner) in each instance of the algorithm. For now, I assume that the set of servers is fixed, so all instances of the consensus algorithm

use the same sets of agents.

In normal operation, a single server is elected to be the leader, which acts as the distinguished proposer (the only one that tries to issue proposals) in all instances of the consensus algorithm. Clients send commands to the leader, who decides where in the sequence each command should appear. If the leader decides that a certain client command should be the 135th command, it tries to have that command chosen as the value of the 135th instance of the consensus algorithm. It will usually succeed. It might fail because of failures, or because another server also believes itself to be the leader and has a different idea of what the 135th command should be. But the consensus algorithm ensures that at most one command can be chosen as the 135th one.

Key to the efficiency of this approach is that, in the Paxos consensus algorithm, the value to be proposed is not chosen until phase 2. Recall that, after completing phase 1 of the proposer's algorithm, either the value to be proposed is determined or else the proposer is free to propose any value.

I will now describe how the Paxos state machine implementation works during normal operation. Later, I will discuss what can go wrong. I consider what happens when the previous leader has just failed and a new leader has been selected. (System startup is a special case in which no commands have yet been proposed.)

The new leader, being a learner in all instances of the consensus algorithm, should know most of the commands that have already been chosen. Suppose it knows commands 1–134, 138, and 139—that is, the values chosen in instances 1–134, 138, and 139 of the consensus algorithm. (We will see later how such a gap in the command sequence could arise.) It then executes phase 1 of instances 135–137 and of all instances greater than 139. (I describe below how this is done.) Suppose that the outcome of these executions determine the value to be proposed in instances 135 and 140, but leaves the proposed value unconstrained in all other instances. The leader then executes phase 2 for instances 135 and 140, thereby choosing commands 135 and 140.

The leader, as well as any other server that learns all the commands the leader knows, can now execute commands 1–135. However, it can't execute commands 138–140, which it also knows, because commands 136 and 137 have yet to be chosen. The leader could take the next two commands requested by clients to be commands 136 and 137. Instead, we let it fill the gap immediately by proposing, as commands 136 and 137, a special "no-op" command that leaves the state unchanged. (It does this by executing phase 2 of instances 136 and 137 of the consensus algorithm.) Once these

no-op commands have been chosen, commands 138–140 can be executed.

Commands 1–140 have now been chosen. The leader has also completed phase 1 for all instances greater than 140 of the consensus algorithm, and it is free to propose any value in phase 2 of those instances. It assigns command number 141 to the next command requested by a client, proposing it as the value in phase 2 of instance 141 of the consensus algorithm. It proposes the next client command it receives as command 142, and so on.

The leader can propose command 142 before it learns that its proposed command 141 has been chosen. It's possible for all the messages it sent in proposing command 141 to be lost, and for command 142 to be chosen before any other server has learned what the leader proposed as command 141. When the leader fails to receive the expected response to its phase 2 messages in instance 141, it will retransmit those messages. If all goes well, its proposed command will be chosen. However, it could fail first, leaving a gap in the sequence of chosen commands. In general, suppose a leader can get α commands ahead—that is, it can propose commands $i+1$ through $i+\alpha$ after commands 1 through i are chosen. A gap of up to $\alpha-1$ commands could then arise.

A newly chosen leader executes phase 1 for infinitely many instances of the consensus algorithm—in the scenario above, for instances 135–137 and all instances greater than 139. Using the same proposal number for all instances, it can do this by sending a single reasonably short message to the other servers. In phase 1, an acceptor responds with more than a simple OK only if it has already received a phase 2 message from some proposer. (In the scenario, this was the case only for instances 135 and 140.) Thus, a server (acting as acceptor) can respond for all instances with a single reasonably short message. Executing these infinitely many instances of phase 1 therefore poses no problem.

Since failure of the leader and election of a new one should be rare events, the effective cost of executing a state machine command—that is, of achieving consensus on the command/value—is the cost of executing only phase 2 of the consensus algorithm. It can be shown that phase 2 of the Paxos consensus algorithm has the minimum possible cost of any algorithm for reaching agreement in the presence of faults [2]. Hence, the Paxos algorithm is essentially optimal.

This discussion of the normal operation of the system assumes that there is always a single leader, except for a brief period between the failure of the current leader and the election of a new one. In abnormal circumstances, the leader election might fail. If no server is acting as leader, then no new commands will be proposed. If multiple servers think they are leaders, then

they can all propose values in the same instance of the consensus algorithm, which could prevent any value from being chosen. However, safety is preserved—two different servers will never disagree on the value chosen as the ith state machine command. Election of a single leader is needed only to ensure progress.

If the set of servers can change, then there must be some way of determining what servers implement what instances of the consensus algorithm. The easiest way to do this is through the state machine itself. The current set of servers can be made part of the state and can be changed with ordinary state-machine commands. We can allow a leader to get α commands ahead by letting the set of servers that execute instance $i + \alpha$ of the consensus algorithm be specified by the state after execution of the ith state machine command. This permits a simple implementation of an arbitrarily sophisticated reconfiguration algorithm.

References

[1] Michael J. Fischer, Nancy Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

[2] Idit Keidar and Sergio Rajsbaum. On the cost of fault-tolerant consensus when there are no faults—a tutorial. TechnicalReport MIT-LCS-TR-821, Laboratory for Computer Science, Massachusetts Institute Technology, Cambridge, MA, 02139, May 2001. also published in SIGACT News 32(2) (June 2001).

[3] Leslie Lamport. The implementation of reliable distributed multiprocess systems. *Computer Networks*, 2:95–114, 1978.

[4] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[5] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.

Chapter 7

Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web

Authors:
Seth Gilbert, Nancy Lynch

Publication year: 2002

Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services

Seth Gilbert* Nancy Lynch*

Abstract

When designing distributed web services, there are three properties that are commonly desired: consistency, availability, and partition tolerance. It is impossible to achieve all three. In this note, we prove this conjecture in the asynchronous network model, and then discuss solutions to this dilemma in the partially synchronous model.

1 Introduction

At PODC 2000, Brewer[1], in an invited talk [2], made the following conjecture: it is impossible for a web service to provide the following three guarantees:

- Consistency
- Availability
- Partition-tolerance

All three of these properties are desirable – and expected – from real-world web services. In this note, we will first discuss what Brewer meant by the conjecture; next we will formalize these concepts and prove the conjecture;

*Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139.

[1]Eric Brewer is a professor at the University of California, Berkeley, and the co-founder and Chief Scientist of Inktomi.

finally, we will describe and attempt to formalize some real-world solutions to this practical difficulty.

Most web services today attempt to provide strongly consistent data. There has been significant research designing ACID[2] databases, and most of the new frameworks for building distributed web services depend on these databases. Interactions with web services are expected to behave in a transactional manner: operations commit or fail in their entirety (atomic), committed transactions are visible to all future transactions (consistent), uncommitted transactions are isolated from each other (isolated), and once a transaction is committed it is permanent (durable). It is clearly important, for example, that billing information and commercial transaction records be handled with this type of strong consistency.

Web services are similarly expected to be highly available. Every request should succeed and receive a response. When a service goes down, it may well create significant real-world problems; the classic example of this is the potential legal difficulties should the E-Trade web site go down. This problem is exacerbated by the fact that a web-site is most likely to be unavailable when it is most needed. The goal of most web services today is to be as available as the network on which they run: if any service on the network is available, then the web service should be accessible.

Finally, on a highly distributed network, it is desirable to provide some amount of fault-tolerance. When some nodes crash or some communication links fail, it is important that the service still perform as expected. One desirable fault tolerance property is the ability to survive a network partitioning into multiple components. In this note we will not consider stopping failures, though in some cases a stopping failure can be modeled as a node existing in its own unique component of a partition.

2 Formal Model

In this section, we will formally define what is meant by the terms *consistent*, *available*, and *partition tolerant*.

2.1 Atomic Data Objects

The most natural way of formalizing the idea of a consistent service is as an atomic data object. Atomic [5], or linearizable [4], consistency is the

[2] **A**tomic, **C**onsistent, **I**solated, **D**urable

condition expected by most web services today.[3] Under this consistency guarantee, there must exist a total order on all operations such that each operation looks as if it were completed at a single instant. This is equivalent to requiring requests of the distributed shared memory to act as if they were executing on a single node, responding to operations one at a time. One important property of an atomic read/write shared memory is that any read operation that begins after a write operation completes must return that value, or the result of a later write operation. This is the consistency guarantee that generally provides the easiest model for users to understand, and is most convenient for those attempting to design a client application that uses the distributed service. See [6] for a more complete definition of atomic consistency.

2.2 Available Data Objects

For a distributed system to be continuously available, every request received by a non-failing node in the system must result in a response.[4] That is, any algorithm used by the service must eventually terminate. In some ways this is a weak definition of availability: it puts no bound on how long the algorithm may run before terminating, and therefore allows unbounded computation. On the other hand, when qualified by the need for partition tolerance, this can be seen as a strong definition of availability: even when severe network failures occur, every request must terminate.

2.3 Partition Tolerance

The above definitions of availability and atomicity are qualified by the need to tolerate partitions. In order to model partition tolerance, the network will be allowed to lose arbitrarily many messages sent from one node to another. When a network is partitioned, all messages sent from nodes in one component of the partition to nodes in another component are lost. (And any pattern of message loss can be modeled as a temporary partition separating the communicating nodes at the exact instant the message is lost.)

[3] Discussing atomic consistency is somewhat different than talking about an ACID database, as database consistency refers to transactions, while atomic consistency refers only to a property of a single request/response operation sequence. And it has a different meaning than the **A**tomic in ACID, as it subsumes the database notions of both **A**tomic and **C**onsistent.

[4] Brewer originally only required *almost all* requests to receive a response. As allowing probabilistic availability does not change the result when arbitrary failures occur, for simplicity we are requiring 100% availability.

The atomicity requirement (§2.1) therefore implies that every response will be atomic, even though arbitrary messages sent as part of the algorithm might not be delivered. The availability requirement (§2.2) implies that every node receiving a request from a client must respond, even though arbitrary messages that are sent may be lost. Note that this is similar to wait-free termination in a pure shared-memory system: even if every other node in the network fails (i.e. the node is in its own unique component of the partition), a valid (atomic) response must be generated. No set of failures less than total network failure is allowed to cause the system to respond incorrectly.[5]

3 Asynchronous Networks

3.1 Impossibility Result

In proving this conjecture, we will use the asynchronous network model, as formalized by Lynch in [7]. In the asynchronous model, there is no clock, and nodes must make decisions based only on the messages received and local computation.

Theorem 1 *It is impossible in the asynchronous network model to implement a read/write data object that guarantees the following properties:*

- *Availability*
- *Atomic consistency*

in all fair executions (including those in which messages are lost).

Proof: We prove this by contradiction. Assume an algorithm A exists that meets the three criteria: atomicity, availability, and partition tolerance. We construct an execution of A in which there exists a request that returns an inconsistent response. The methodology is similar to proofs in Attiya et al. [1] and Lynch [8]. Assume that the network consists of at least two nodes. Thus it can be divided into two disjoint, non-empty sets: $\{G_1, G_2\}$. The basic idea of the proof is to assume that all messages between G_1 and G_2 are lost. Then if a *write* occurs in G_1, and later a *read* occurs in G_2, then the *read* operation cannot return the results of the earlier *write* operation.

[5] Brewer pointed out in the talk that partitions of one node are irrelevant: they are equivalent to that node failing. However restricting our attention to partitions containing only components of size greater than one does not change any of the results in this note.

More formally, let v_0 be the initial value of the atomic object. Let α_1 be the prefix of an execution of A in which a single *write* of a value not equal to v_0 occurs in G_1, ending with the termination of the *write* operation. Assume that no other client requests occur in either G_1 or G_2. Further, assume that no messages from G_1 are received in G_2, and no messages from G_2 are received in G_1. We know that this *write* completes, by the availability requirement. Similarly, let α_2 be the prefix of an execution in which a single *read* occurs in G_2, and no other client requests occur, ending with the termination of the *read* operation. During α_2 no messages from G_2 are received in G_1, and no messages from G_1 are received in G_2. Again we know that the *read* returns a value by the availability requirement. The value returned by this execution must be v_0, as no *write* operation has occurred in α_2.

Let α be an execution beginning with α_1 and continuing with α_2. To the nodes in G_2, α is indistinguishable from α_2, as all the messages from G_1 to G_2 are lost (in both α_1 and α_2, which together make up α), and α_1 does not include any client requests to nodes in G_2. Therefore in the α execution, the *read* request (from α_2) must still return v_0. However the *read* request does not begin until after the *write* request (from α_1) has completed. This therefore contradicts the atomicity property, proving that no such algorithm exists. ■

Corollary 1.1 *It is impossible in the asynchronous network model to implement a read/write data object that guarantees the following properties:*

- *Availability, in all fair executions,*
- *Atomic consistency, in fair executions in which* no *messages are lost.*

Proof: The main idea is that in the asynchronous model an algorithm has no way of determining whether a message has been lost, or has been arbitrarily delayed in the transmission channel. Therefore if there existed an algorithm that guaranteed atomic consistency in executions in which no messages were lost, then there would exist an algorithm that guaranteed atomic consistency in all executions. This would violate Theorem 1.

More formally, assume for the sake of contradiction that there exists an algorithm A that always terminates, and guarantees atomic consistency in fair executions in which all messages are delivered. Further, Theorem 1 implies that A does not guarantee atomic consistency in all fair executions, so there exists some fair execution α of A in which some response is not atomic.

At some finite point in execution α, the algorithm A returns a response that is not atomic. Let α' be the prefix of α ending with the invalid response. Next, extend α' to a fair execution α'', in which all messages are delivered. The execution α'' is now a fair execution in which all messages are delivered. However this execution is not atomic. Therefore no such algorithm A exists. ■

3.2 Solutions in the Asynchronous Model

While it is impossible to provide all three properties: atomicity, availability, and partition tolerance, any two of these three properties can be achieved.

3.2.1 Atomic, Partition Tolerant

If availability is not required, then it is easy to achieve atomic data and partition tolerance. The trivial system that ignores all requests meets these requirements. However we can provide a stronger liveness criterion: if all the messages in an execution are delivered, the system is available and all operations terminate. A simple centralized algorithm meets these requirements: a single designated node maintains the value of an object. A node receiving a request forwards the request to the designated node, which sends a response. When an acknowledgment is received, the node sends a response to the client.

Many distributed databases provide this type of guarantee, especially algorithms based on distributed locking or quorums: if certain failure patterns occur, then the liveness condition is weakened and the service no longer returns responses. If there are no failures, then liveness is guaranteed.

3.2.2 Atomic, Available

If there are no partitions, it is clearly possible to provide atomic, available data. In fact, the centralized algorithm described in Section 3.2.1 meets these requirements. Systems that run on intranets and LANs are an example of these types of algorithms.

3.2.3 Available, Partition Tolerant

It is possible to provide high availability and partition tolerance, if atomic consistency is not required. If there are no consistency requirements, the service can trivially return v_0, the initial value, in response to every request.

However it is possible to provide weakened consistency in an available, partition tolerant setting. Web caches are one example of a weakly consistent network. In Section 4.3 we consider some of the weaker consistency conditions that are possible.

4 Partially Synchronous Networks

4.1 Partially Synchronous Model

The most obvious way to try to circumvent the impossibility result of Theorem 1 is to realize that in the real world, most networks are not purely asynchronous. If you allow each node in the network to have a clock, it is possible to build a more powerful service.

For the rest of this paper, we will discuss a partially synchronous model in which every node has a clock, and all clocks increase at the same rate. However, the clocks themselves are not synchronized, in that they may display different values at the same real time. In effect, the clocks act as timers: local state variables that the processes can observe to measure how much time has passed. A local timer can be used to schedule an action to occur a certain interval of time after some other event. Furthermore, assume that every message is either delivered within a given, known time: t_{msg}, or it is lost. Also, every node processes a received message within a given, known time: t_{local}, and local processing takes zero time. This can be formalized as a special case of the General Timed Automata model defined by Lynch [9].

4.2 Impossibility Result

It is still impossible to have an always available, atomic data object when arbitrary messages may be lost, even in the partially synchronous model. That is, the following analogue of Theorem 1 holds:

Theorem 2 *It is impossible in the partially synchronous network model to implement a read/write data object that guarantees the following properties:*

- *Availability*
- *Atomic consistency*

in all executions (even those in which messages are lost).

Proof: This proof is rather similar to the proof of Theorem 1. We will follow the same methodology: divide the network into two components, $\{G_1, G_2\}$,

and construct an admissable execution in which a *write* happens in one component, followed by a *read* operation in the other component. This *read* operation can be shown to return inconsistent data.

More formally, construct execution α_1 as before in Theorem 1: a single *write* request and acknowledgment occur in G_1, and all messages between the two components, $\{G_1, G_2\}$, are lost. We will construct the second execution, α_2', slightly differently. Let α_2' be an execution that begins with a long interval of time during which no client requests occur. This interval must be at least as long as the entire duration of α_1. Then append to α_2' the events of α_2, as defined above in Theorem 1: a single *read* request and response in G_2, again assuming all messages between the two components are lost. Finally, construct α by superimposing the two executions α_1 and α_2'. The long interval of time in α_2 ensures that the *write* request completes before the *read* request begins. However, as in Theorem 1, the *read* request returns the initial value, rather than the new value written by the *write* request, violating atomic consistency. ■

4.3 Solutions in the Partially Synchronous Model

In the partially synchronous model, however, the analogue of Corollary 1.1 does not hold. The proof of this corollary does in fact depend on nodes being unaware of when a message is lost. There are partially synchronous algorithms that will return atomic data when all messages in an execution are delivered (i.e., there are no partitions), and will only return inconsistent (and, in particular, stale) data when messages are lost. One example of such an algorithm is the centralized protocol described in Section 3.2.1, modified to time-out lost messages. On a *read* (or *write*) request, a message is sent to the central node. If a response from the central node is received, then the node delivers the requested data (or an acknowledgment). If no response is received within $2 * t_{msg} + t_{local}$, then the node concludes that the message was lost. The client is then sent a response: either the best known value of the local node, or an acknowledgment. In this case, atomic consistency may be violated.

4.4 Weaker Consistency Conditions

While it is useful to guarantee that atomic data will be returned in executions in which all messages are delivered (within some time bound), it is equally important to specify what happens in executions in which some of the messages are lost. In this section, we will discuss one possible weaker con-

sistency condition that would allow stale data to be returned when there are partitions, yet still place formal requirements on the quality of the stale data returned. This consistency guarantee will require availability and atomic consistency in executions in which no messages are lost, and is therefore impossible to guarantee in the asynchronous model as a result of Corollary 1.1.

In the partially synchronous model it often makes sense to base guarantees on how long an algorithm has had to rectify a situation. This consistency model ensures that if messages are delivered, then eventually some notion of atomicity is restored.

In an atomic execution, we would define a partial order of the *read* and *write* operations, and then require that if one operation begins after another one ends, the former does not precede the latter in the partial order. We will define a weaker guarantee, Delayed-t consistency, that defines a partial order in a similar manner, but only requires that one operation not precede another if there was an interval between the operations in which all messages were delivered.

Definition 3 *A timed execution, α, of a read-write object is* Delayed-t Consistent *if:*

1. *P is a partial order that orders all* write *operations, and orders all* read *operations with respect to the* write *operations.*

2. *The value returned by every* read *operation is exactly the one written by the previous* write *operation in P (or the initial value, if there is no such previous* write *in P).*

3. *The order in P is consistent with the order of* read *and* write *requests submitted at each node.*

4. *(Atomicity) If all messages in the execution are delivered, and an operation θ completes before an operation ϕ begins, then ϕ does not precede θ in the partial order P,*

5. *(Weakly Consistent) Assume there exists an interval of time longer than t in which no messages are lost. Further, assume an operation, θ, completes before the interval begins, and another operation, ϕ, begins after the interval ends. Then ϕ does not precede θ in the partial order P.*

This guarantee allows for some stale data when messages are lost, but provides a time limit on how long this inconsistency can continue, once the partition heals. This is related to the idea of eventually consistent data, as described in Fekete et al. [3]; here, however, we want an explicit time bound on how long it will take for the data to become consistent.

A variant of the centralized algorithm described in Section 4.3 is Delayed-t consistent. Assume node C is the centralized node. The algorithm behaves as follows:

- *read* at node A:
  1. A sends a request to C for the most recent value.
  2. If A receives a response from C, save the value and send it to the client.
  3. If A concludes that a message was lost (i.e. a timeout occurs), then return the value with the highest sequence number received from C (see below), or the initial-value (if no value has yet been received from C).
- *write* at A:
  1. A sends a message to C with the new value.
  2. If A receives an acknowledgement from C, then A sends an acknowledgement to the client, and stops.
  3. If A concludes a message was lost (i.e. a timeout occurs), then A sends an acknowledgement to the client.
  4. If A has not yet received an acknowledgement from C, then A sends a message to C with the new value.
  5. If A concludes a message was lost (i.e. a timeout occurs), A repeats step 4 within $t - 4 * t_{\text{timeout}}$ seconds.
- New value is received at C:
  1. C increments its sequence number by 1.
  2. C sends out the new value and the sequence number to every node.
  3. If C concludes a message was lost (i.e. a timeout occurs), then C resends the value and sequence number to the missing node within time $t - 2 * t_{\text{timeout}}$ seconds.

4. Repeat step 3 until every node has acknowledged the value.

Theorem 4 *The modified centralized algorithm is Delayed-t consistent.*

Proof: The ordering of *write* operations is the order in which the centralized node is notified of the client *write* operations. The centralized node assigns each a sequence number, and this determines the total ordering of *write* operations. Each *read* operation is sequenced after the *write* operation whose value it returns. If all the messages in an execution are delivered, this algorithm is atomic: the centralized node serializes requests, and ensures that the partial order is correctly respected. If an operation has completed, then the centralized node has already been notified, and therefore no operation that begins later will precede the completed operation in the partial order.

If, on the other hand, some messages are lost, the resulting executions might not be atomic. Assume a *write* occurs at A_w, after which an interval of time passes in which all messages are delivered. By the end of the interval, A_w will have notified the central node of the new value, and the central node will have notified all other nodes of this new value. Therefore no later operation after this interval will precede the *write* operation.

Assume, then, that a *read* operation occurs at B_r, after which an interval of time passes in which all messages are delivered. B_r must have received its value from the central node, C, or from an earlier *write* operation at B_r. In the former case, by the end of this interval C will have ensured that every node has received a value at least as recent as the one returned by B_r. In the latter case, by the end of the interval B_r will have sent the value to C and C will have forwarded it to every other node. Therefore no later operation after this interval will precede the *read* operation. ■

5 Conclusion

In this note, we have shown that it is impossible to reliably provide atomic, consistent data when there are partitions in the network. It is feasible, however, to achieve any two of the three properties: consistency, availability, and partition tolerance. In an asynchronous model, when no clocks are available, the impossibility result is fairly strong: it is impossible to provide consistent data, even allowing stale data to be returned when messages are lost. However in partially synchronous models it is possible to achieve a practical compromise between consistency and availability. In particular, most real-world systems today are forced to settle with returning "most of

the data, most of the time." Formalizing this idea and studying algorithms for achieving it is an interesting subject for future theoretical research.

References

[1] Attiya, Bar-Noy, Dolev, Koller, Peleg, and Reischuk. Achievable cases in an asynchronous environment. In *28th Annual Symposium on Foundations of Computer Science*, pages 337–346, Los Angeles, California, October 1987.

[2] Eric A. Brewer. Towards robust distributed systems. (Invited Talk) *Principles of Distributed Computing*, Portland, Oregon, July 2000.

[3] Alan Fekete, David Gupta, Victor Luchangco, Nancy Lynch, and Alex Shvartsman. Eventually-serializable data services. *Theoretical Computer Science*, 220(1):113–156, June 1999.

[4] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[5] Leslie Lamport. On interprocess communication – parts i and ii. *Distributed Computing*, 1(2):77–101, April 1986.

[6] Nancy Lynch. *Distributed Algorithms*, pages 397–350. Morgan Kaufman, 1996.

[7] Nancy Lynch. *Distributed Algorithms*, pages 199–231. Morgan Kaufman, 1996.

[8] Nancy Lynch. *Distributed Algorithms*, page 581. Morgan Kaufman, 1996.

[9] Nancy Lynch. *Distributed Algorithms*, pages 735–770. Morgan Kaufman, 1996.

Chapter 8

Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications

Authors:
Avinash Lakshman, Prashant Malik

Publication year: 2010

Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications

Ion Stoica*, Robert Morris, David Karger, M. Frans Kaashoek, Hari Balakrishnan†
MIT Laboratory for Computer Science
chord@lcs.mit.edu
http://pdos.lcs.mit.edu/chord/

Abstract

A fundamental problem that confronts peer-to-peer applications is to efficiently locate the node that stores a particular data item. This paper presents *Chord*, a distributed lookup protocol that addresses this problem. Chord provides support for just one operation: given a key, it maps the key onto a node. Data location can be easily implemented on top of Chord by associating a key with each data item, and storing the key/data item pair at the node to which the key maps. Chord adapts efficiently as nodes join and leave the system, and can answer queries even if the system is continuously changing. Results from theoretical analysis, simulations, and experiments show that Chord is scalable, with communication cost and the state maintained by each node scaling logarithmically with the number of Chord nodes.

1. Introduction

Peer-to-peer systems and applications are distributed systems without any centralized control or hierarchical organization, where the software running at each node is equivalent in functionality. A review of the features of recent peer-to-peer applications yields a long list: redundant storage, permanence, selection of nearby servers, anonymity, search, authentication, and hierarchical naming. Despite this rich set of features, the core operation in most peer-to-peer systems is efficient location of data items. The contribution of this paper is a scalable protocol for lookup in a dynamic peer-to-peer system with frequent node arrivals and departures.

The *Chord protocol* supports just one operation: given a key, it maps the key onto a node. Depending on the application using Chord, that node might be responsible for storing a value associated with the key. Chord uses a variant of consistent hashing [11] to assign keys to Chord nodes. Consistent hashing tends to balance load, since each node receives roughly the same number of keys, and involves relatively little movement of keys when nodes join and leave the system.

Previous work on consistent hashing assumed that nodes were aware of most other nodes in the system, making it impractical to scale to large number of nodes. In contrast, each Chord node needs "routing" information about only a few other nodes. Because the routing table is distributed, a node resolves the hash function by communicating with a few other nodes. In the steady state, in an N-node system, each node maintains information only about $O(\log N)$ other nodes, and resolves all lookups via $O(\log N)$ messages to other nodes. Chord maintains its routing information as nodes join and leave the system; with high probability each such event results in no more than $O(\log^2 N)$ messages.

Three features that distinguish Chord from many other peer-to-peer lookup protocols are its simplicity, provable correctness, and provable performance. Chord is simple, routing a key through a sequence of $O(\log N)$ other nodes toward the destination. A Chord node requires information about $O(\log N)$ other nodes for *efficient* routing, but performance degrades gracefully when that information is out of date. This is important in practice because nodes will join and leave arbitrarily, and consistency of even $O(\log N)$ state may be hard to maintain. Only one piece information per node need be correct in order for Chord to guarantee correct (though slow) routing of queries; Chord has a simple algorithm for maintaining this information in a dynamic environment.

The rest of this paper is structured as follows. Section 2 compares Chord to related work. Section 3 presents the system model that motivates the Chord protocol. Section 4 presents the base Chord protocol and proves several of its properties, while Section 5 presents extensions to handle concurrent joins and failures. Section 6 demonstrates our claims about Chord's performance through simulation and experiments on a deployed prototype. Finally, we outline items for future work in Section 7 and summarize our contributions in Section 8.

2. Related Work

While Chord maps keys onto nodes, traditional name and location services provide a *direct* mapping between keys and values. A value can be an address, a document, or an arbitrary data item. Chord can easily implement this functionality by storing each key/value pair at the node to which that key maps. For this reason and to make the comparison clearer, the rest of this section assumes a Chord-based service that maps keys onto values.

DNS provides a host name to IP address mapping [15]. Chord can provide the same service with the name representing the key and the associated IP address representing the value. Chord requires no special servers, while DNS relies on a set of special root

*University of California, Berkeley. istoica@cs.berkeley.edu
†Authors in reverse alphabetical order.

This research was sponsored by the Defense Advanced Research Projects Agency (DARPA) and the Space and Naval Warfare Systems Center, San Diego, under contract N66001-00-1-8933.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGCOMM'01, August 27-31, 2001, San Diego, California, USA.
Copyright 2001 ACM 1-58113-411-8/01/0008 ...$5.00.

servers. DNS names are structured to reflect administrative boundaries; Chord imposes no naming structure. DNS is specialized to the task of finding named hosts or services, while Chord can also be used to find data objects that are not tied to particular machines.

The Freenet peer-to-peer storage system [4, 5], like Chord, is decentralized and symmetric and automatically adapts when hosts leave and join. Freenet does not assign responsibility for documents to specific servers; instead, its lookups take the form of searches for cached copies. This allows Freenet to provide a degree of anonymity, but prevents it from guaranteeing retrieval of existing documents or from providing low bounds on retrieval costs. Chord does not provide anonymity, but its lookup operation runs in predictable time and always results in success or definitive failure.

The Ohaha system uses a consistent hashing-like algorithm for mapping documents to nodes, and Freenet-style query routing [18]. As a result, it shares some of the weaknesses of Freenet. Archival Intermemory uses an off-line computed tree to map logical addresses to machines that store the data [3].

The Globe system [2] has a wide-area location service to map object identifiers to the locations of moving objects. Globe arranges the Internet as a hierarchy of geographical, topological, or administrative domains, effectively constructing a static world-wide search tree, much like DNS. Information about an object is stored in a particular leaf domain, and pointer caches provide search short cuts [22]. The Globe system handles high load on the logical root by partitioning objects among multiple physical root servers using hash-like techniques. Chord performs this hash function well enough that it can achieve scalability without also involving any hierarchy, though Chord does not exploit network locality as well as Globe.

The distributed data location protocol developed by Plaxton *et al.* [19], a variant of which is used in OceanStore [12], is perhaps the closest algorithm to the Chord protocol. It provides stronger guarantees than Chord: like Chord it guarantees that queries make a logarithmic number hops and that keys are well balanced, but the Plaxton protocol also ensures, subject to assumptions about network topology, that queries never travel further in network distance than the node where the key is stored. The advantage of Chord is that it is substantially less complicated and handles concurrent node joins and failures well. The Chord protocol is also similar to Pastry, the location algorithm used in PAST [8]. However, Pastry is a prefix-based routing protocol, and differs in other details from Chord.

CAN uses a d-dimensional Cartesian coordinate space (for some fixed d) to implement a distributed hash table that maps keys onto values [20]. Each node maintains $O(d)$ state, and the lookup cost is $O(dN^{1/d})$. Thus, in contrast to Chord, the state maintained by a CAN node does not depend on the network size N, but the lookup cost increases faster than $\log N$. If $d = \log N$, CAN lookup times and storage needs match Chord's. However, CAN is not designed to vary d as N (and thus $\log N$) varies, so this match will only occur for the "right" N corresponding to the fixed d. CAN requires an additional maintenance protocol to periodically remap the identifier space onto nodes. Chord also has the advantage that its correctness is robust in the face of partially incorrect routing information.

Chord's routing procedure may be thought of as a one-dimensional analogue of the Grid location system [14]. Grid relies on real-world geographic location information to route its queries; Chord maps its nodes to an artificial one-dimensional space within which routing is carried out by an algorithm similar to Grid's.

Chord can be used as a lookup service to implement a variety of systems, as discussed in Section 3. In particular, it can help avoid single points of failure or control that systems like Napster possess [17], and the lack of scalability that systems like Gnutella display because of their widespread use of broadcasts [10].

3. System Model

Chord simplifies the design of peer-to-peer systems and applications based on it by addressing these difficult problems:

- **Load balance:** Chord acts as a distributed hash function, spreading keys evenly over the nodes; this provides a degree of natural load balance.
- **Decentralization:** Chord is fully distributed: no node is more important than any other. This improves robustness and makes Chord appropriate for loosely-organized peer-to-peer applications.
- **Scalability:** The cost of a Chord lookup grows as the log of the number of nodes, so even very large systems are feasible. No parameter tuning is required to achieve this scaling.
- **Availability:** Chord automatically adjusts its internal tables to reflect newly joined nodes as well as node failures, ensuring that, barring major failures in the underlying network, the node responsible for a key can always be found. This is true even if the system is in a continuous state of change.
- **Flexible naming:** Chord places no constraints on the structure of the keys it looks up: the Chord key-space is flat. This gives applications a large amount of flexibility in how they map their own names to Chord keys.

The Chord software takes the form of a library to be linked with the client and server applications that use it. The application interacts with Chord in two main ways. First, Chord provides a `lookup(key)` algorithm that yields the IP address of the node responsible for the key. Second, the Chord software on each node notifies the application of changes in the set of keys that the node is responsible for. This allows the application software to, for example, move corresponding values to their new homes when a new node joins.

The application using Chord is responsible for providing any desired authentication, caching, replication, and user-friendly naming of data. Chord's flat key space eases the implementation of these features. For example, an application could authenticate data by storing it under a Chord key derived from a cryptographic hash of the data. Similarly, an application could replicate data by storing it under two distinct Chord keys derived from the data's application-level identifier.

The following are examples of applications for which Chord would provide a good foundation:

Cooperative Mirroring, as outlined in a recent proposal [6]. Imagine a set of software developers, each of whom wishes to publish a distribution. Demand for each distribution might vary dramatically, from very popular just after a new release to relatively unpopular between releases. An efficient approach for this would be for the developers to cooperatively mirror each others' distributions. Ideally, the mirroring system would balance the load across all servers, replicate and cache the data, and ensure authenticity. Such a system should be fully decentralized in the interests of reliability, and because there is no natural central administration.

Time-Shared Storage for nodes with intermittent connectivity. If a person wishes some data to be always available, but their

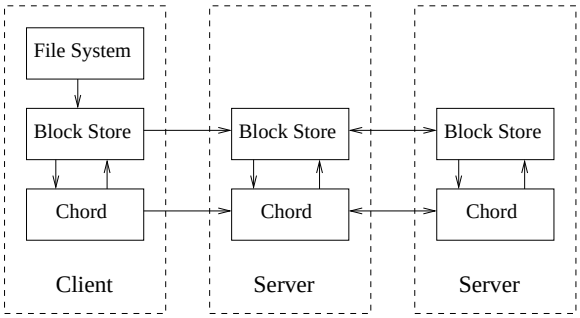


Figure 1: Structure of an example Chord-based distributed storage system.

machine is only occasionally available, they can offer to store others' data while they are up, in return for having their data stored elsewhere when they are down. The data's name can serve as a key to identify the (live) Chord node responsible for storing the data item at any given time. Many of the same issues arise as in the Cooperative Mirroring application, though the focus here is on availability rather than load balance.

Distributed Indexes to support Gnutella- or Napster-like keyword search. A key in this application could be derived from the desired keywords, while values could be lists of machines offering documents with those keywords.

Large-Scale Combinatorial Search, such as code breaking. In this case keys are candidate solutions to the problem (such as cryptographic keys); Chord maps these keys to the machines responsible for testing them as solutions.

Figure 1 shows a possible three-layered software structure for a cooperative mirror system. The highest layer would provide a file-like interface to users, including user-friendly naming and authentication. This "file system" layer might implement named directories and files, mapping operations on them to lower-level block operations. The next layer, a "block storage" layer, would implement the block operations. It would take care of storage, caching, and replication of blocks. The block storage layer would use Chord to identify the node responsible for storing a block, and then talk to the block storage server on that node to read or write the block.

4. The Base Chord Protocol

The Chord protocol specifies how to find the locations of keys, how new nodes join the system, and how to recover from the failure (or planned departure) of existing nodes. This section describes a simplified version of the protocol that does not handle concurrent joins or failures. Section 5 describes enhancements to the base protocol to handle concurrent joins and failures.

4.1 Overview

At its heart, Chord provides fast distributed computation of a hash function mapping keys to nodes responsible for them. It uses *consistent hashing* [11, 13], which has several good properties. With high probability the hash function balances load (all nodes receive roughly the same number of keys). Also with high probability, when an Nth node joins (or leaves) the network, only an $O(1/N)$ fraction of the keys are moved to a different location—this is clearly the minimum necessary to maintain a balanced load.

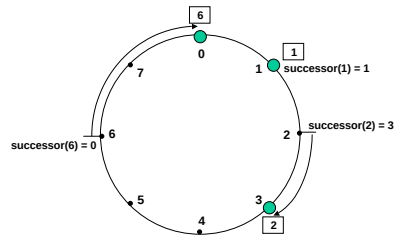


Figure 2: An identifier circle consisting of the three nodes 0, 1, and 3. In this example, key 1 is located at node 1, key 2 at node 3, and key 6 at node 0.

Chord improves the scalability of consistent hashing by avoiding the requirement that every node know about every other node. A Chord node needs only a small amount of "routing" information about other nodes. Because this information is distributed, a node resolves the hash function by communicating with a few other nodes. In an N-node network, each node maintains information only about $O(\log N)$ other nodes, and a lookup requires $O(\log N)$ messages.

Chord must update the routing information when a node joins or leaves the network; a join or leave requires $O(\log^2 N)$ messages.

4.2 Consistent Hashing

The consistent hash function assigns each node and key an m-bit *identifier* using a base hash function such as SHA-1 [9]. A node's identifier is chosen by hashing the node's IP address, while a key identifier is produced by hashing the key. We will use the term "key" to refer to both the original key and its image under the hash function, as the meaning will be clear from context. Similarly, the term "node" will refer to both the node and its identifier under the hash function. The identifier length m must be large enough to make the probability of two nodes or keys hashing to the same identifier negligible.

Consistent hashing assigns keys to nodes as follows. Identifiers are ordered in an *identifier circle* modulo 2^m. Key k is assigned to the first node whose identifier is equal to or follows (the identifier of) k in the identifier space. This node is called the *successor node* of key k, denoted by *successor*(k). If identifiers are represented as a circle of numbers from 0 to $2^m - 1$, then $successor(k)$ is the first node clockwise from k.

Figure 2 shows an identifier circle with $m = 3$. The circle has three nodes: 0, 1, and 3. The successor of identifier 1 is node 1, so key 1 would be located at node 1. Similarly, key 2 would be located at node 3, and key 6 at node 0.

Consistent hashing is designed to let nodes enter and leave the network with minimal disruption. To maintain the consistent hashing mapping when a node n joins the network, certain keys previously assigned to n's successor now become assigned to n. When node n leaves the network, all of its assigned keys are reassigned to n's successor. No other changes in assignment of keys to nodes need occur. In the example above, if a node were to join with identifier 7, it would capture the key with identifier 6 from the node with identifier 0.

The following results are proven in the papers that introduced consistent hashing [11, 13]:

THEOREM 1. *For any set of N nodes and K keys, with high probability:*

1. *Each node is responsible for at most $(1+\epsilon)K/N$ keys*

2. *When an $(N+1)^{st}$ node joins or leaves the network, responsibility for $O(K/N)$ keys changes hands (and only to or from the joining or leaving node).*

When consistent hashing is implemented as described above, the theorem proves a bound of $\epsilon = O(\log N)$. The consistent hashing paper shows that ϵ can be reduced to an arbitrarily small constant by having each node run $O(\log N)$ "virtual nodes" each with its own identifier.

The phrase "with high probability" bears some discussion. A simple interpretation is that the nodes and keys are randomly chosen, which is plausible in a non-adversarial model of the world. The probability distribution is then over random choices of keys and nodes, and says that such a random choice is unlikely to produce an unbalanced distribution. One might worry, however, about an adversary who intentionally chooses keys to all hash to the same identifier, destroying the load balancing property. The consistent hashing paper uses "k-universal hash functions" to provide certain guarantees even in the case of nonrandom keys.

Rather than using a k-universal hash function, we chose to use the standard SHA-1 function as our base hash function. This makes our protocol deterministic, so that the claims of "high probability" no longer make sense. However, producing a set of keys that collide under SHA-1 can be seen, in some sense, as inverting, or "decrypting" the SHA-1 function. This is believed to be hard to do. Thus, instead of stating that our theorems hold with high probability, we can claim that they hold "based on standard hardness assumptions."

For simplicity (primarily of presentation), we dispense with the use of virtual nodes. In this case, the load on a node may exceed the average by (at most) an $O(\log N)$ factor with high probability (or in our case, based on standard hardness assumptions). One reason to avoid virtual nodes is that the number needed is determined by the number of nodes in the system, which may be difficult to determine. Of course, one may choose to use an a priori upper bound on the number of nodes in the system; for example, we could postulate at most one Chord server per IPv4 address. In this case running 32 virtual nodes per physical node would provide good load balance.

4.3 Scalable Key Location

A very small amount of routing information suffices to implement consistent hashing in a distributed environment. Each node need only be aware of its successor node on the circle. Queries for a given identifier can be passed around the circle via these successor pointers until they first encounter a node that succeeds the identifier; this is the node the query maps to. A portion of the Chord protocol maintains these successor pointers, thus ensuring that all lookups are resolved correctly. However, this resolution scheme is inefficient: it may require traversing all N nodes to find the appropriate mapping. To accelerate this process, Chord maintains additional routing information. This additional information is not essential for correctness, which is achieved as long as the successor information is maintained correctly.

As before, let m be the number of bits in the key/node identifiers. Each node, n, maintains a routing table with (at most) m entries, called the *finger table*. The ith entry in the table at node n contains the identity of the *first* node, s, that succeeds n by at least 2^{i-1} on the identifier circle, i.e., $s = successor(n + 2^{i-1})$, where $1 \leq i \leq m$ (and all arithmetic is modulo 2^m). We call node s the ith *finger* of node n, and denote it by $n.finger[i].node$ (see Table 1). A finger table entry includes both the Chord identifier and the IP address (and port number) of the relevant node. Note that the first finger of n is its immediate successor on the circle; for convenience we often refer to it as the *successor* rather than the first finger.

In the example shown in Figure 3(b), the finger table of node 1 points to the successor nodes of identifiers $(1 + 2^0) \bmod 2^3 = 2$, $(1 + 2^1) \bmod 2^3 = 3$, and $(1 + 2^2) \bmod 2^3 = 5$, respectively. The successor of identifier 2 is node 3, as this is the first node that follows 2, the successor of identifier 3 is (trivially) node 3, and the successor of 5 is node 0.

Notation	Definition
finger$[k].start$	$(n + 2^{k-1}) \bmod 2^m$, $1 \leq k \leq m$
.interval	[*finger*$[k]$.*start*, *finger*$[k+1]$.*start*)
$.node$	first node $\geq n.$*finger*$[k]$.*start*
successor	the next node on the identifier circle; *finger*[1].*node*
predecessor	the previous node on the identifier circle

Table 1: Definition of variables for node n, using m-bit identifiers.

This scheme has two important characteristics. First, each node stores information about only a small number of other nodes, and knows more about nodes closely following it on the identifier circle than about nodes farther away. Second, a node's finger table generally does not contain enough information to determine the successor of an arbitrary key k. For example, node 3 in Figure 3 does not know the successor of 1, as 1's successor (node 1) does not appear in node 3's finger table.

What happens when a node n does not know the successor of a key k? If n can find a node whose ID is closer than its own to k, that node will know more about the identifier circle in the region of k than n does. Thus n searches its finger table for the node j whose ID most immediately precedes k, and asks j for the node it knows whose ID is closest to k. By repeating this process, n learns about nodes with IDs closer and closer to k.

The pseudocode that implements the search process is shown in Figure 4. The notation *n.foo()* stands for the function *foo()* being invoked at and executed on node n. Remote calls and variable references are preceded by the remote node identifier, while local variable references and procedure calls omit the local node. Thus *n.foo()* denotes a remote procedure call on node n, while *n.bar*, without parentheses, is an RPC to lookup a variable *bar* on node n.

find_successor works by finding the immediate predecessor node of the desired identifier; the successor of that node must be the successor of the identifier. We implement *find_predecessor* explicitly, because it is used later to implement the join operation (Section 4.4).

When node n executes *find_predecessor*, it contacts a series of nodes moving forward around the Chord circle towards id. If node n contacts a node n' such that id falls between n' and the successor of n', *find_predecessor* is done and returns n'. Otherwise node n asks n' for the node n' knows about that most closely precedes id. Thus the algorithm always makes progress towards the precedessor of id.

As an example, consider the Chord ring in Figure 3(b). Suppose node 3 wants to find the successor of identifier 1. Since 1 belongs to the circular interval $[7, 3)$, it belongs to 3.*finger*[3].*interval*; node 3 therefore checks the third entry in its finger table, which is 0. Because 0 precedes 1, node 3 will ask node 0 to find the successor of 1. In turn, node 0 will infer from its finger table that 1's successor is the node 1 itself, and return node 1 to node 3.

The finger pointers at repeatedly doubling distances around the circle cause each iteration of the loop in *find_predecessor* to halve the distance to the target identifier. From this intuition follows a theorem:

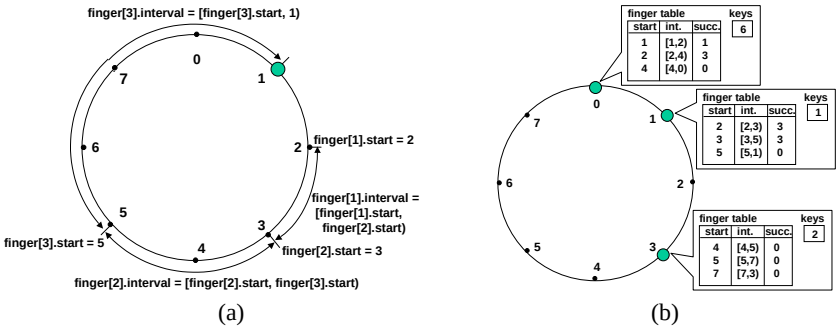


Figure 3: (a) The finger intervals associated with node 1. (b) Finger tables and key locations for a net with nodes 0, 1, and 3, and keys 1, 2, and 6.

THEOREM 2. *With high probability (or under standard hardness assumptions), the number of nodes that must be contacted to find a successor in an N-node network is $O(\log N)$.*

PROOF. Suppose that node n wishes to resolve a query for the successor of k. Let p be the node that immediately precedes k. We analyze the number of query steps to reach p.

Recall that if $n \neq p$, then n forwards its query to the closest predecessor of k in its finger table. Suppose that node p is in the ith finger interval of node n. Then since this interval is not empty, node n will finger some node f in this interval. The distance (number of identifiers) between n and f is at least 2^{i-1}. But f and p are both in n's i^{th} finger interval, which means the distance between them is at most 2^{i-1}. This means f is closer to p than to n, or equivalently, that the distance from f to p is at most half the distance from n to p.

If the distance between the node handling the query and the predecessor p halves in each step, and is at most 2^m initially, then within m steps the distance will be one, meaning we have arrived at p.

In fact, as discussed above, we assume that node and key identifiers are random. In this case, the number of forwardings necessary will be $O(\log N)$ with high probability. After $\log N$ forwardings, the distance between the current query node and the key k will be reduced to at most $2^m/N$. The expected number of node identifiers landing in a range of this size is 1, and it is $O(\log N)$ with high probability. Thus, even if the remaining steps advance by only one node at a time, they will cross the entire remaining interval and reach key k within another $O(\log N)$ steps. □

In the section reporting our experimental results (Section 6), we will observe (and justify) that the average lookup time is $\frac{1}{2}\log N$.

4.4 Node Joins

In a dynamic network, nodes can join (and leave) at any time. The main challenge in implementing these operations is preserving the ability to locate every key in the network. To achieve this goal, Chord needs to preserve two invariants:

1. Each node's successor is correctly maintained.
2. For every key k, node $successor(k)$ is responsible for k.

In order for lookups to be fast, it is also desirable for the finger tables to be correct.

This section shows how to maintain these invariants when a single node joins. We defer the discussion of multiple nodes joining simultaneously to Section 5, which also discusses how to handle

```
// ask node n to find id's successor
n.find_successor(id)
  n' = find_predecessor(id);
  return n'.successor;

// ask node n to find id's predecessor
n.find_predecessor(id)
  n' = n;
  while (id ∉ (n', n'.successor])
    n' = n'.closest_preceding_finger(id);
  return n';

// return closest finger preceding id
n.closest_preceding_finger(id)
  for i = m downto 1
    if (finger[i].node ∈ (n, id))
      return finger[i].node;
  return n;
```

Figure 4: The pseudocode to find the successor node of an identifier id. Remote procedure calls and variable lookups are preceded by the remote node.

a node failure. Before describing the join operation, we summarize its performance (the proof of this theorem is in the companion technical report [21]):

THEOREM 3. *With high probability, any node joining or leaving an N-node Chord network will use $O(\log^2 N)$ messages to re-establish the Chord routing invariants and finger tables.*

To simplify the join and leave mechanisms, each node in Chord maintains a *predecessor pointer*. A node's predecessor pointer contains the Chord identifier and IP address of the immediate predecessor of that node, and can be used to walk counterclockwise around the identifier circle.

To preserve the invariants stated above, Chord must perform three tasks when a node n joins the network:

1. Initialize the predecessor and fingers of node n.
2. Update the fingers and predecessors of existing nodes to reflect the addition of n.
3. Notify the higher layer software so that it can transfer state (e.g. values) associated with keys that node n is now responsible for.

We assume that the new node learns the identity of an existing Chord node n' by some external mechanism. Node n uses n' to

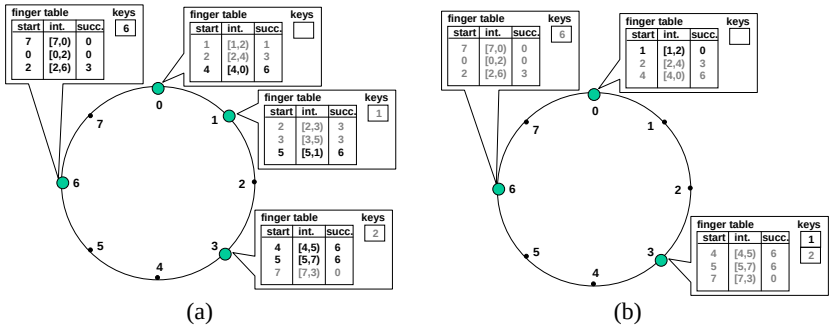


Figure 5: (a) Finger tables and key locations after node 6 joins. (b) Finger tables and key locations after node 3 leaves. Changed entries are shown in black, and unchanged in gray.

initialize its state and add itself to the existing Chord network, as follows.

Initializing fingers and predecessor: Node n learns its predecessor and fingers by asking n' to look them up, using the *init_finger_table* pseudocode in Figure 6. Naively performing *find_successor* for each of the m finger entries would give a run-time of $O(m \log N)$. To reduce this, n checks whether the ith finger is also the correct $(i+1)^{th}$ finger, for each i. This happens when *finger*$[i]$.*interval* does not contain any node, and thus *finger*$[i]$.*node* $\geq$ *finger*$[i+1]$.*start*. It can be shown that the change reduces the expected (and high probability) number of finger entries that must be looked up to $O(\log N)$, which reduces the overall time to $O(\log^2 N)$.

As a practical optimization, a newly joined node n can ask an immediate neighbor for a copy of its complete finger table and its predecessor. n can use the contents of these tables as hints to help it find the correct values for its own tables, since n's tables will be similar to its neighbors'. This can be shown to reduce the time to fill the finger table to $O(\log N)$.

Updating fingers of existing nodes: Node n will need to be entered into the finger tables of some existing nodes. For example, in Figure 5(a), node 6 becomes the third finger of nodes 0 and 1, and the first and the second finger of node 3.

Figure 6 shows the pseudocode of the *update_finger_table* function that updates existing finger tables. Node n will become the ith finger of node p if and only if (1) p precedes n by at least 2^{i-1}, and (2) the ith finger of node p succeeds n. The first node, p, that can meet these two conditions is the immediate predecessor of $n - 2^{i-1}$. Thus, for a given n, the algorithm starts with the ith finger of node n, and then continues to walk in the counter-clock-wise direction on the identifier circle until it encounters a node whose i^{th} finger precedes n.

We show in the technical report [21] that the number of nodes that need to be updated when a node joins the network is $O(\log N)$ with high probability. Finding and updating these nodes takes $O(\log^2 N)$ time. A more sophisticated scheme can reduce this time to $O(\log N)$; however, we do not present it as we expect implementations to use the algorithm of the following section.

Transferring keys: The last operation that has to be performed when a node n joins the network is to move responsibility for all the keys for which node n is now the successor. Exactly what this entails depends on the higher-layer software using Chord, but typically it would involve moving the data associated with each key to the new node. Node n can become the successor only for keys that were previously the responsibility of the node immediately follow-

```
#define successor finger[1].node

// node n joins the network;
// n' is an arbitrary node in the network
n.join(n')
  if (n')
    init_finger_table(n');
    update_others();
    // move keys in (predecessor, n] from successor
  else // n is the only node in the network
    for i = 1 to m
      finger[i].node = n;
    predecessor = n;

// initialize finger table of local node;
// n' is an arbitrary node already in the network
n.init_finger_table(n')
  finger[1].node = n'.find_successor(finger[1].start);
  predecessor = successor.predecessor;
  successor.predecessor = n;
  for i = 1 to m - 1
    if (finger[i + 1].start ∈ [n, finger[i].node))
      finger[i + 1].node = finger[i].node;
    else
      finger[i + 1].node =
          n'.find_successor(finger[i + 1].start);


// update all nodes whose finger
// tables should refer to n
n.update_others()
  for i = 1 to m
    // find last node p whose i^th finger might be n
    p = find_predecessor(n - 2^(i-1));
    p.update_finger_table(n, i);

// if s is i^th finger of n, update n's finger table with s
n.update_finger_table(s, i)
  if (s ∈ [n, finger[i].node))
    finger[i].node = s;
    p = predecessor; // get first node preceding n
    p.update_finger_table(s, i);
```

Figure 6: Pseudocode for the node join operation.

ing n, so n only needs to contact that one node to transfer responsibility for all relevant keys.

5. Concurrent Operations and Failures

In practice Chord needs to deal with nodes joining the system concurrently and with nodes that fail or leave voluntarily. This section describes modifications to the basic Chord algorithms described in Section 4 to handle these situations.

5.1 Stabilization

The join algorithm in Section 4 aggressively maintains the finger tables of all nodes as the network evolves. Since this invariant is difficult to maintain in the face of concurrent joins in a large network, we separate our correctness and performance goals. A basic "stabilization" protocol is used to keep nodes' successor pointers up to date, which is sufficient to guarantee correctness of lookups. Those successor pointers are then used to verify and correct finger table entries, which allows these lookups to be fast as well as correct.

If joining nodes have affected some region of the Chord ring, a lookup that occurs before stabilization has finished can exhibit one of three behaviors. The common case is that all the finger table entries involved in the lookup are reasonably current, and the lookup finds the correct successor in $O(\log N)$ steps. The second case is where successor pointers are correct, but fingers are inaccurate. This yields correct lookups, but they may be slower. In the final case, the nodes in the affected region have incorrect successor pointers, or keys may not yet have migrated to newly joined nodes, and the lookup may fail. The higher-layer software using Chord will notice that the desired data was not found, and has the option of retrying the lookup after a pause. This pause can be short, since stabilization fixes successor pointers quickly.

Our stabilization scheme guarantees to add nodes to a Chord ring in a way that preserves reachability of existing nodes, even in the face of concurrent joins and lost and reordered messages. Stabilization by itself won't correct a Chord system that has split into multiple disjoint cycles, or a single cycle that loops multiple times around the identifier space. These pathological cases cannot be produced by any sequence of ordinary node joins. It is unclear whether they can be produced by network partitions and recoveries or intermittent failures. If produced, these cases could be detected and repaired by periodic sampling of the ring topology.

Figure 7 shows the pseudo-code for joins and stabilization; this code replaces Figure 6 to handle concurrent joins. When node n first starts, it calls $n.join(n')$, where n' is any known Chord node. The $join$ function asks n' to find the immediate successor of n. By itself, $join$ does not make the rest of the network aware of n.

Every node runs *stabilize* periodically (this is how newly joined nodes are noticed by the network). When node n runs *stabilize*, it asks n's successor for the successor's predecessor p, and decides whether p should be n's successor instead. This would be the case if node p recently joined the system. *stabilize* also notifies node n's successor of n's existence, giving the successor the chance to change its predecessor to n. The successor does this only if it knows of no closer predecessor than n.

As a simple example, suppose node n joins the system, and its ID lies between nodes n_p and n_s. n would acquire n_s as its successor. Node n_s, when notified by n, would acquire n as its predecessor. When n_p next runs *stabilize*, it will ask n_s for its predecessor (which is now n); n_p would then acquire n as its successor. Finally, n_p will notify n, and n will acquire n_p as its predecessor. At this point, all predecessor and successor pointers are correct.

```
n.join(n')
    predecessor = nil;
    successor = n'.find_successor(n);

// periodically verify n's immediate successor,
// and tell the successor about n.
n.stabilize()
    x = successor.predecessor;
    if (x ∈ (n, successor))
        successor = x;
    successor.notify(n);

// n' thinks it might be our predecessor.
n.notify(n')
    if (predecessor is nil or n' ∈ (predecessor, n))
        predecessor = n';

// periodically refresh finger table entries.
n.fix_fingers()
    i = random index > 1 into finger[];
    finger[i].node = find_successor(finger[i].start);
```

Figure 7: Pseudocode for stabilization.

As soon as the successor pointers are correct, calls to *find_predecessor* (and thus *find_successor*) will work. Newly joined nodes that have not yet been fingered may cause *find_predecessor* to initially undershoot, but the loop in the lookup algorithm will nevertheless follow successor (*finger*[1]) pointers through the newly joined nodes until the correct predecessor is reached. Eventually *fix_fingers* will adjust finger table entries, eliminating the need for these linear scans.

The following theorems (proved in the technical report [21]) show that all problems caused by concurrent joins are transient. The theorems assume that any two nodes trying to communicate will eventually succeed.

THEOREM 4. *Once a node can successfully resolve a given query, it will always be able to do so in the future.*

THEOREM 5. *At some time after the last join all successor pointers will be correct.*

The proofs of these theorems rely on an invariant and a termination argument. The invariant states that once node n can reach node r via successor pointers, it always can. To argue termination, we consider the case where two nodes both think they have the same successor s. In this case, each will attempt to notify s, and s will eventually choose the closer of the two (or some other, closer node) as its predecessor. At this point the farther of the two will, by contacting s, learn of a better successor than s. It follows that every node progresses towards a better and better successor over time. This progress must eventually halt in a state where every node is considered the successor of exactly one other node; this defines a cycle (or set of them, but the invariant ensures that there will be at most one).

We have not discussed the adjustment of fingers when nodes join because it turns out that joins don't substantially damage the performance of fingers. If a node has a finger into each interval, then these fingers can still be used even after joins. The distance halving argument is essentially unchanged, showing that $O(\log N)$ hops suffice to reach a node "close" to a query's target. New joins influence the lookup only by getting in between the old predecessor and successor of a target query. These new nodes may need to be scanned linearly (if their fingers are not yet accurate). But unless a

tremendous number of nodes joins the system, the number of nodes between two old nodes is likely to be very small, so the impact on lookup is negligible. Formally, we can state the following:

THEOREM 6. *If we take a stable network with N nodes, and another set of up to N nodes joins the network with no finger pointers (but with correct successor pointers), then lookups will still take $O(\log N)$ time with high probability.*

More generally, so long as the time it takes to adjust fingers is less than the time it takes the network to double in size, lookups should continue to take $O(\log N)$ hops.

5.2 Failures and Replication

When a node n fails, nodes whose finger tables include n must find n's successor. In addition, the failure of n must not be allowed to disrupt queries that are in progress as the system is re-stabilizing.

The key step in failure recovery is maintaining correct successor pointers, since in the worst case *find_predecessor* can make progress using only successors. To help achieve this, each Chord node maintains a "successor-list" of its r nearest successors on the Chord ring. In ordinary operation, a modified version of the *stabilize* routine in Figure 7 maintains the successor-list. If node n notices that its successor has failed, it replaces it with the first live entry in its successor list. At that point, n can direct ordinary lookups for keys for which the failed node was the successor to the new successor. As time passes, *stabilize* will correct finger table entries and successor-list entries pointing to the failed node.

After a node failure, but before stabilization has completed, other nodes may attempt to send requests through the failed node as part of a *find_successor* lookup. Ideally the lookups would be able to proceed, after a timeout, by another path despite the failure. In many cases this is possible. All that is needed is a list of alternate nodes, easily found in the finger table entries preceding that of the failed node. If the failed node had a very low finger table index, nodes in the successor-list are also available as alternates.

The technical report proves the following two theorems that show that the successor-list allows lookups to succeed, and be efficient, even during stabilization [21]:

THEOREM 7. *If we use a successor list of length $r = O(\log N)$ in a network that is initially stable, and then every node fails with probability 1/2, then with high probability find_successor returns the closest living successor to the query key.*

THEOREM 8. *If we use a successor list of length $r = O(\log N)$ in a network that is initially stable, and then every node fails with probability 1/2, then the expected time to execute find_successor in the failed network is $O(\log N)$.*

The intuition behind these proofs is straightforward: a node's r successors all fail with probability $2^{-r} = 1/N$, so with high probability a node will be aware of, so able to forward messages to, its closest living successor.

The successor-list mechanism also helps higher layer software replicate data. A typical application using Chord might store replicas of the data associated with a key at the k nodes succeeding the key. The fact that a Chord node keeps track of its r successors means that it can inform the higher layer software when successors come and go, and thus when the software should propagate new replicas.

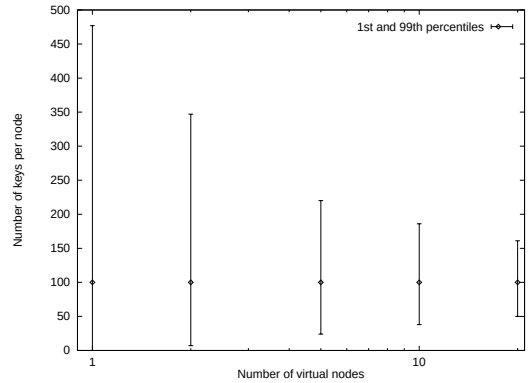


Figure 9: The 1st and the 99th percentiles of the number of keys per node as a function of virtual nodes mapped to a real node. The network has 10^4 real nodes and stores 10^6 keys.

6. Simulation and Experimental Results

In this section, we evaluate the Chord protocol by simulation. The simulator uses the lookup algorithm in Figure 4 and a slightly older version of the stabilization algorithms described in Section 5. We also report on some preliminary experimental results from an operational Chord-based system running on Internet hosts.

6.1 Protocol Simulator

The Chord protocol can be implemented in an *iterative* or *recursive* style. In the iterative style, a node resolving a lookup initiates all communication: it asks a series of nodes for information from their finger tables, each time moving closer on the Chord ring to the desired successor. In the recursive style, each intermediate node forwards a request to the next node until it reaches the successor. The simulator implements the protocols in an iterative style.

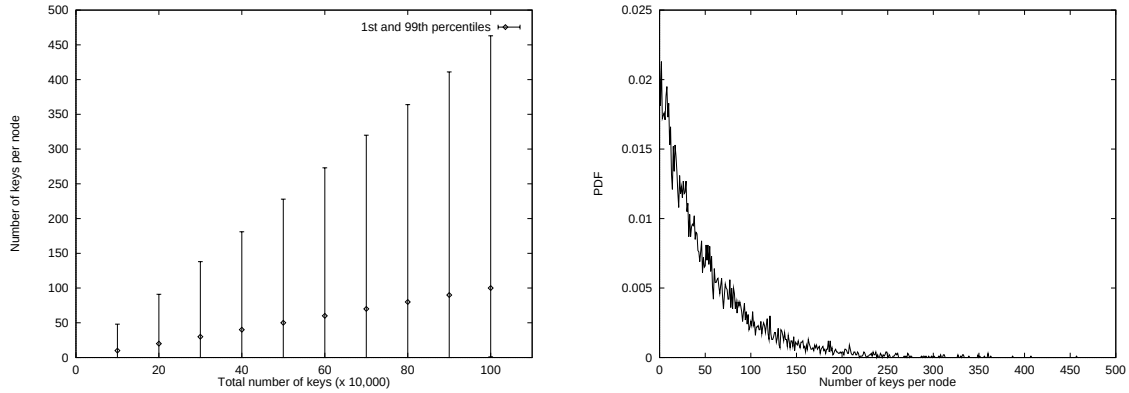
6.2 Load Balance

We first consider the ability of consistent hashing to allocate keys to nodes evenly. In a network with N nodes and K keys we would like the distribution of keys to nodes to be tight around N/K.

We consider a network consisting of 10^4 nodes, and vary the total number of keys from 10^5 to 10^6 in increments of 10^5. For each value, we repeat the experiment 20 times. Figure 8(a) plots the mean and the 1st and 99th percentiles of the number of keys per node. The number of keys per node exhibits large variations that increase linearly with the number of keys. For example, in all cases some nodes store no keys. To clarify this, Figure 8(b) plots the probability density function (PDF) of the number of keys per node when there are 5×10^5 keys stored in the network. The maximum number of nodes stored by any node in this case is 457, or $9.1\times$ the mean value. For comparison, the 99th percentile is $4.6\times$ the mean value.

One reason for these variations is that node identifiers do not uniformly cover the entire identifier space. If we divide the identifier space in N equal-sized bins, where N is the number of nodes, then we might hope to see one node in each bin. But in fact, the probability that a particular bin does not contain any node is $(1-1/N)^N$. For large values of N this approaches $e^{-1} = 0.368$.

As we discussed earlier, the consistent hashing paper solves this problem by associating keys with virtual nodes, and mapping multiple virtual nodes (with unrelated identifiers) to each real node. Intuitively, this will provide a more uniform coverage of the identifier space. For example, if we allocate $\log N$ randomly chosen virtual nodes to each real node, with high probability each of the



(a) (b)

Figure 8: (a) The mean and 1st and 99th percentiles of the number of keys stored per node in a 10^4 node network. (b) The probability density function (PDF) of the number of keys per node. The total number of keys is 5×10^5.

N bins will contain $O(\log N)$ nodes [16]. We note that this does not affect the worst-case query path length, which now becomes $O(\log(N \log N)) = O(\log N)$.

To verify this hypothesis, we perform an experiment in which we allocate r virtual nodes to each real node. In this case keys are associated to virtual nodes instead of real nodes. We consider again a network with 10^4 real nodes and 10^6 keys. Figure 9 shows the 1st and 99th percentiles for $r = 1, 2, 5, 10$, and 20, respectively. As expected, the 99th percentile decreases, while the 1st percentile increases with the number of virtual nodes, r. In particular, the 99th percentile decreases from $4.8\times$ to $1.6\times$ the mean value, while the 1st percentile increases from 0 to $0.5\times$ the mean value. Thus, adding virtual nodes as an indirection layer can significantly improve load balance. The tradeoff is that routing table space usage will increase as each actual node now needs r times as much space to store the finger tables for its virtual nodes. However, we believe that this increase can be easily accommodated in practice. For example, assuming a network with $N = 10^6$ nodes, and assuming $r = \log N$, each node has to maintain a table with only $\log^2 N \simeq 400$ entries.

6.3 Path Length

The performance of any routing protocol depends heavily on the length of the path between two arbitrary nodes in the network. In the context of Chord, we define the path length as the number of nodes traversed during a lookup operation. From Theorem 2, with high probability, the length of the path to resolve a query is $O(\log N)$, where N is the total number of nodes in the network.

To understand Chord's routing performance in practice, we simulated a network with $N = 2^k$ nodes, storing 100×2^k keys in all. We varied k from 3 to 14 and conducted a separate experiment for each value. Each node in an experiment picked a random set of keys to query from the system, and we measured the path length required to resolve each query.

Figure 10(a) plots the mean, and the 1st and 99th percentiles of path length as a function of k. As expected, the mean path length increases logarithmically with the number of nodes, as do the 1st and 99th percentiles. Figure 10(b) plots the PDF of the path length for a network with 2^{12} nodes ($k = 12$).

Figure 10(a) shows that the path length is about $\frac{1}{2}\log_2 N$. The reason for the $\frac{1}{2}$ is as follows. Consider some random node and a random query. Let the distance in identifier space be considered in binary representation. The most significant (say i^{th}) bit of this distance can be corrected to 0 by following the node's i^{th} finger. If the next significant bit of the distance is 1, it too needs to be corrected by following a finger, but if it is 0, then no $i-1^{st}$ finger is followed—instead, we move on the the $i-2^{nd}$ bit. In general, the number of fingers we need to follow will be the number of ones in the binary representation of the distance from node to query. Since the distance is random, we expect half the $\log N$ bits to be ones.

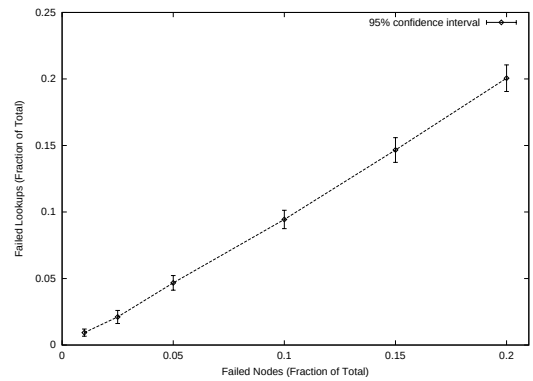
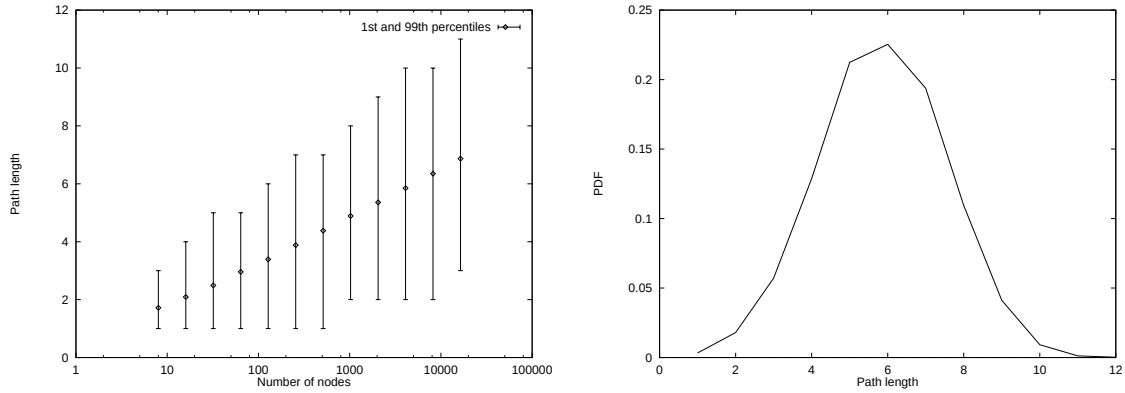


Figure 11: The fraction of lookups that fail as a function of the fraction of nodes that fail.

6.4 Simultaneous Node Failures

In this experiment, we evaluate the ability of Chord to regain consistency after a large percentage of nodes fail simultaneously. We consider again a 10^4 node network that stores 10^6 keys, and randomly select a fraction p of nodes that fail. After the failures occur, we wait for the network to finish stabilizing, and then measure the fraction of keys that could not be looked up correctly. A correct lookup of a key is one that finds the node that was originally responsible for the key, before the failures; this corresponds to a system that stores values with keys but does not replicate the values or recover them after failures.

Figure 11 plots the mean lookup failure rate and the 95% confidence interval as a function of p. The lookup failure rate is almost exactly p. Since this is just the fraction of keys expected to be lost due to the failure of the responsible nodes, we conclude that there is no significant lookup failure in the Chord network. For example, if the Chord network had partitioned in two equal-sized halves, we



(a) (b)

Figure 10: (a) The path length as a function of network size. (b) The PDF of the path length in the case of a 2^{12} node network.

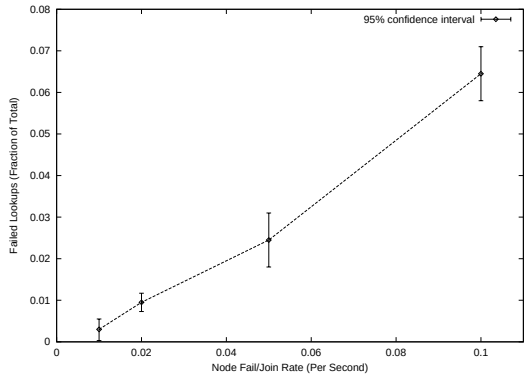


Figure 12: The fraction of lookups that fail as a function of the rate (over time) at which nodes fail and join. Only failures caused by Chord state inconsistency are included, not failures due to lost keys.

would expect one-half of the requests to fail because the querier and target would be in different partitions half the time. Our results do not show this, suggesting that Chord is robust in the face of multiple simultaneous node failures.

6.5 Lookups During Stabilization

A lookup issued after some failures but before stabilization has completed may fail for two reasons. First, the node responsible for the key may have failed. Second, some nodes' finger tables and predecessor pointers may be inconsistent due to concurrent joins and node failures. This section evaluates the impact of continuous joins and failures on lookups.

In this experiment, a lookup is considered to have succeeded if it reaches the current successor of the desired key. This is slightly optimistic: in a real system, there might be periods of time in which the real successor of a key has not yet acquired the data associated with the key from the previous successor. However, this method allows us to focus on Chord's ability to perform lookups, rather than on the higher-layer software's ability to maintain consistency of its own data. Any query failure will be the result of inconsistencies in Chord. In addition, the simulator does not retry queries: if a query is forwarded to a node that is down, the query simply fails. Thus, the results given in this section can be viewed as the worst-case scenario for the query failures induced by state inconsistency.

Because the primary source of inconsistencies is nodes joining and leaving, and because the main mechanism to resolve these inconsistencies is the stabilize protocol, Chord's performance will be sensitive to the frequency of node joins and leaves versus the frequency at which the stabilization protocol is invoked.

In this experiment, key lookups are generated according to a Poisson process at a rate of one per second. Joins and failures are modeled by a Poisson process with the mean arrival rate of R. Each node runs the stabilization routines at randomized intervals averaging 30 seconds; unlike the routines in Figure 7, the simulator updates all finger table entries on every invocation. The network starts with 500 nodes.

Figure 12 plots the average failure rates and confidence intervals. A node failure rate of 0.01 corresponds to one node joining and leaving every 100 seconds on average. For comparison, recall that each node invokes the stabilize protocol once every 30 seconds. In other words, the graph x axis ranges from a rate of 1 failure per 3 stabilization steps to a rate of 3 failures per one stabilization step. The results presented in Figure 12 are averaged over approximately two hours of simulated time. The confidence intervals are computed over 10 independent runs.

The results of figure 12 can be explained roughly as follows. The simulation has 500 nodes, meaning lookup path lengths average around 5. A lookup fails if its finger path encounters a failed node. If k nodes fail, the probability that one of them is on the finger path is roughly $5k/500$, or $k/100$. This would suggest a failure rate of about 3% if we have 3 failures between stabilizations. The graph shows results in this ball-park, but slightly worse since it might take more than one stabilization to completely clear out a failed node.

6.6 Experimental Results

This section presents latency measurements obtained from a prototype implementation of Chord deployed on the Internet. The Chord nodes are at ten sites on a subset of the RON test-bed in the United States [1], in California, Colorado, Massachusetts, New York, North Carolina, and Pennsylvania. The Chord software runs on UNIX, uses 160-bit keys obtained from the SHA-1 cryptographic hash function, and uses TCP to communicate between nodes. Chord runs in the iterative style. These Chord nodes are part of an experimental distributed file system [7], though this section considers only the Chord component of the system.

Figure 13 shows the measured latency of Chord lookups over a range of numbers of nodes. Experiments with a number of nodes larger than ten are conducted by running multiple independent

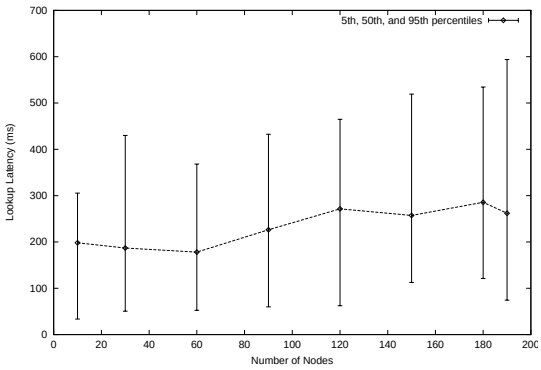


Figure 13: Lookup latency on the Internet prototype, as a function of the total number of nodes. Each of the ten physical sites runs multiple independent copies of the Chord node software.

copies of the Chord software at each site. This is different from running $O(\log N)$ virtual nodes at each site to provide good load balance; rather, the intention is to measure how well our implementation scales even though we do not have more than a small number of deployed nodes.

For each number of nodes shown in Figure 13, each physical site issues 16 Chord lookups for randomly chosen keys one-by-one. The graph plots the median, the 5th, and the 95th percentile of lookup latency. The median latency ranges from 180 to 285 ms, depending on number of nodes. For the case of 180 nodes, a typical lookup involves five two-way message exchanges: four for the Chord lookup, and a final message to the successor node. Typical round-trip delays between sites are 60 milliseconds (as measured by `ping`). Thus the expected lookup time for 180 nodes is about 300 milliseconds, which is close to the measured median of 285. The low 5th percentile latencies are caused by lookups for keys close (in ID space) to the querying node and by query hops that remain local to the physical site. The high 95th percentiles are caused by lookups whose hops follow high delay paths.

The lesson from Figure 13 is that lookup latency grows slowly with the total number of nodes, confirming the simulation results that demonstrate Chord's scalability.

7. Future Work

Based on our experience with the prototype mentioned in Section 6.6, we would like to improve the Chord design in the following areas.

Chord currently has no specific mechanism to heal partitioned rings; such rings could appear locally consistent to the stabilization procedure. One way to check global consistency is for each node n to periodically ask other nodes to do a Chord lookup for n; if the lookup does not yield node n, there may be a partition. This will only detect partitions whose nodes know of each other. One way to obtain this knowledge is for every node to know of the same small set of initial nodes. Another approach might be for nodes to maintain long-term memory of a random set of nodes they have encountered in the past; if a partition forms, the random sets in one partition are likely to include nodes from the other partition.

A malicious or buggy set of Chord participants could present an incorrect view of the Chord ring. Assuming that the data Chord is being used to locate is cryptographically authenticated, this is a threat to availability of data rather than to authenticity. The same approach used above to detect partitions could help victims realize that they are not seeing a globally consistent view of the Chord ring.

An attacker could target a particular data item by inserting a node into the Chord ring with an ID immediately following the item's key, and having the node return errors when asked to retrieve the data. Requiring (and checking) that nodes use IDs derived from the SHA-1 hash of their IP addresses makes this attack harder.

Even $\log N$ messages per lookup may be too many for some applications of Chord, especially if each message must be sent to a random Internet host. Instead of placing its fingers at distances that are all powers of 2, Chord could easily be changed to place its fingers at distances that are all integer powers of $1 + 1/d$. Under such a scheme, a single routing hop could decrease the distance to a query to $1/(1+d)$ of the original distance, meaning that $\log_{1+d} N$ hops would suffice. However, the number of fingers needed would increase to $\log N/(\log(1 + 1/d) \approx O(d \log N)$.

A different approach to improving lookup latency might be to use server selection. Each finger table entry could point to the first k nodes in that entry's interval on the ID ring, and a node could measure the network delay to each of the k nodes. The k nodes are generally equivalent for purposes of lookup, so a node could forward lookups to the one with lowest delay. This approach would be most effective with recursive Chord lookups, in which the node measuring the delays is also the node forwarding the lookup.

8. Conclusion

Many distributed peer-to-peer applications need to determine the node that stores a data item. The Chord protocol solves this challenging problem in decentralized manner. It offers a powerful primitive: given a key, it determines the node responsible for storing the key's value, and does so efficiently. In the steady state, in an N-node network, each node maintains routing information for only about $O(\log N)$ other nodes, and resolves all lookups via $O(\log N)$ messages to other nodes. Updates to the routing information for nodes leaving and joining require only $O(\log^2 N)$ messages.

Attractive features of Chord include its simplicity, provable correctness, and provable performance even in the face of concurrent node arrivals and departures. It continues to function correctly, albeit at degraded performance, when a node's information is only partially correct. Our theoretical analysis, simulations, and experimental results confirm that Chord scales well with the number of nodes, recovers from large numbers of simultaneous node failures and joins, and answers most lookups correctly even during recovery.

We believe that Chord will be a valuable component for peer-to-peer, large-scale distributed applications such as cooperative file sharing, time-shared available storage systems, distributed indices for document and service discovery, and large-scale distributed computing platforms.

Acknowledgments

We thank Frank Dabek for the measurements of the Chord prototype described in Section 6.6, and David Andersen for setting up the testbed used in those measurements.

9. References

[1] ANDERSEN, D. Resilient overlay networks. Master's thesis, Department of EECS, MIT, May 2001. `http://nms.lcs.mit.edu/projects/ron/`.

[2] BAKKER, A., AMADE, E., BALLINTIJN, G., KUZ, I., VERKAIK, P., VAN DER WIJK, I., VAN STEEN, M., AND TANENBAUM., A.

The Globe distribution network. In *Proc. 2000 USENIX Annual Conf. (FREENIX Track)* (San Diego, CA, June 2000), pp. 141–152.

[3] CHEN, Y., EDLER, J., GOLDBERG, A., GOTTLIEB, A., SOBTI, S., AND YIANILOS, P. A prototype implementation of archival intermemory. In *Proceedings of the 4th ACM Conference on Digital libraries* (Berkeley, CA, Aug. 1999), pp. 28–37.

[4] CLARKE, I. A distributed decentralised information storage and retrieval system. Master's thesis, University of Edinburgh, 1999.

[5] CLARKE, I., SANDBERG, O., WILEY, B., AND HONG, T. W. Freenet: A distributed anonymous information storage and retrieval system. In *Proceedings of the ICSI Workshop on Design Issues in Anonymity and Unobservability* (Berkeley, California, June 2000). `http://freenet.sourceforge.net`.

[6] DABEK, F., BRUNSKILL, E., KAASHOEK, M. F., KARGER, D., MORRIS, R., STOICA, I., AND BALAKRISHNAN, H. Building peer-to-peer systems with Chord, a distributed location service. In *Proceedings of the 8th IEEE Workshop on Hot Topics in Operating Systems (HotOS-VIII)* (Elmau/Oberbayern, Germany, May 2001), pp. 71–76.

[7] DABEK, F., KAASHOEK, M. F., KARGER, D., MORRIS, R., AND STOICA, I. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)* (To appear; Banff, Canada, Oct. 2001).

[8] DRUSCHEL, P., AND ROWSTRON, A. Past: Persistent and anonymous storage in a peer-to-peer networking environment. In *Proceedings of the 8th IEEE Workshop on Hot Topics in Operating Systems (HotOS 2001)* (Elmau/Oberbayern, Germany, May 2001), pp. 65–70.

[9] FIPS 180-1. *Secure Hash Standard*. U.S. Department of Commerce/NIST, National Technical Information Service, Springfield, VA, Apr. 1995.

[10] Gnutella. `http://gnutella.wego.com/`.

[11] KARGER, D., LEHMAN, E., LEIGHTON, F., LEVINE, M., LEWIN, D., AND PANIGRAHY, R. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In *Proceedings of the 29th Annual ACM Symposium on Theory of Computing* (El Paso, TX, May 1997), pp. 654–663.

[12] KUBIATOWICZ, J., BINDEL, D., CHEN, Y., CZERWINSKI, S., EATON, P., GEELS, D., GUMMADI, R., RHEA, S., WEATHERSPOON, H., WEIMER, W., WELLS, C., AND ZHAO, B. OceanStore: An architecture for global-scale persistent storage. In *Proceeedings of the Ninth international Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000)* (Boston, MA, November 2000), pp. 190–201.

[13] LEWIN, D. Consistent hashing and random trees: Algorithms for caching in distributed networks. Master's thesis, Department of EECS, MIT, 1998. Available at the MIT Library, `http://thesis.mit.edu/`.

[14] LI, J., JANNOTTI, J., DE COUTO, D., KARGER, D., AND MORRIS, R. A scalable location service for geographic ad hoc routing. In *Proceedings of the 6th ACM International Conference on Mobile Computing and Networking* (Boston, Massachusetts, August 2000), pp. 120–130.

[15] MOCKAPETRIS, P., AND DUNLAP, K. J. Development of the Domain Name System. In *Proc. ACM SIGCOMM* (Stanford, CA, 1988), pp. 123–133.

[16] MOTWANI, R., AND RAGHAVAN, P. *Randomized Algorithms*. Cambridge University Press, New York, NY, 1995.

[17] Napster. `http://www.napster.com/`.

[18] Ohaha, Smart decentralized peer-to-peer sharing. `http://www.ohaha.com/design.html`.

[19] PLAXTON, C., RAJARAMAN, R., AND RICHA, A. Accessing nearby copies of replicated objects in a distributed environment. In *Proceedings of the ACM SPAA* (Newport, Rhode Island, June 1997), pp. 311–320.

[20] RATNASAMY, S., FRANCIS, P., HANDLEY, M., KARP, R., AND SHENKER, S. A scalable content-addressable network. In *Proc. ACM SIGCOMM* (San Diego, CA, August 2001).

[21] STOICA, I., MORRIS, R., KARGER, D., KAASHOEK, M. F., AND BALAKRISHNAN, H. Chord: A scalable peer-to-peer lookup service for internet applications. Tech. Rep. TR-819, MIT LCS, March 2001. `http://www.pdos.lcs.mit.edu/chord/papers/`.

[22] VAN STEEN, M., HAUCK, F., BALLINTIJN, G., AND TANENBAUM, A. Algorithmic design of the Globe wide-area location service. *The Computer Journal 41*, 5 (1998), 297–310.

Chapter 9

Bigtable: A Distributed Storage System for Structured Data

Authors:
Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach Mike Burrows, Tushar Chandra, Andrew Fikes, Robert E. Gruber

Publication year: 2006

Bigtable: A Distributed Storage System for Structured Data

Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach
Mike Burrows, Tushar Chandra, Andrew Fikes, Robert E. Gruber
{fay,jeff,sanjay,wilsonh,kerr,m3b,tushar,fikes,gruber}@google.com

Google, Inc.

Abstract

Bigtable is a distributed storage system for managing structured data that is designed to scale to a very large size: petabytes of data across thousands of commodity servers. Many projects at Google store data in Bigtable, including web indexing, Google Earth, and Google Finance. These applications place very different demands on Bigtable, both in terms of data size (from URLs to web pages to satellite imagery) and latency requirements (from backend bulk processing to real-time data serving). Despite these varied demands, Bigtable has successfully provided a flexible, high-performance solution for all of these Google products. In this paper we describe the simple data model provided by Bigtable, which gives clients dynamic control over data layout and format, and we describe the design and implementation of Bigtable.

1 Introduction

Over the last two and a half years we have designed, implemented, and deployed a distributed storage system for managing structured data at Google called Bigtable. Bigtable is designed to reliably scale to petabytes of data and thousands of machines. Bigtable has achieved several goals: wide applicability, scalability, high performance, and high availability. Bigtable is used by more than sixty Google products and projects, including Google Analytics, Google Finance, Orkut, Personalized Search, Writely, and Google Earth. These products use Bigtable for a variety of demanding workloads, which range from throughput-oriented batch-processing jobs to latency-sensitive serving of data to end users. The Bigtable clusters used by these products span a wide range of configurations, from a handful to thousands of servers, and store up to several hundred terabytes of data.

In many ways, Bigtable resembles a database: it shares many implementation strategies with databases. Parallel databases [14] and main-memory databases [13] have achieved scalability and high performance, but Bigtable provides a different interface than such systems. Bigtable does not support a full relational data model; instead, it provides clients with a simple data model that supports dynamic control over data layout and format, and allows clients to reason about the locality properties of the data represented in the underlying storage. Data is indexed using row and column names that can be arbitrary strings. Bigtable also treats data as uninterpreted strings, although clients often serialize various forms of structured and semi-structured data into these strings. Clients can control the locality of their data through careful choices in their schemas. Finally, Bigtable schema parameters let clients dynamically control whether to serve data out of memory or from disk.

Section 2 describes the data model in more detail, and Section 3 provides an overview of the client API. Section 4 briefly describes the underlying Google infrastructure on which Bigtable depends. Section 5 describes the fundamentals of the Bigtable implementation, and Section 6 describes some of the refinements that we made to improve Bigtable's performance. Section 7 provides measurements of Bigtable's performance. We describe several examples of how Bigtable is used at Google in Section 8, and discuss some lessons we learned in designing and supporting Bigtable in Section 9. Finally, Section 10 describes related work, and Section 11 presents our conclusions.

2 Data Model

A Bigtable is a sparse, distributed, persistent multi-dimensional sorted map. The map is indexed by a row key, column key, and a timestamp; each value in the map is an uninterpreted array of bytes.

$$(\text{row:string}, \text{column:string}, \text{time:int64}) \rightarrow \text{string}$$

To appear in OSDI 2006

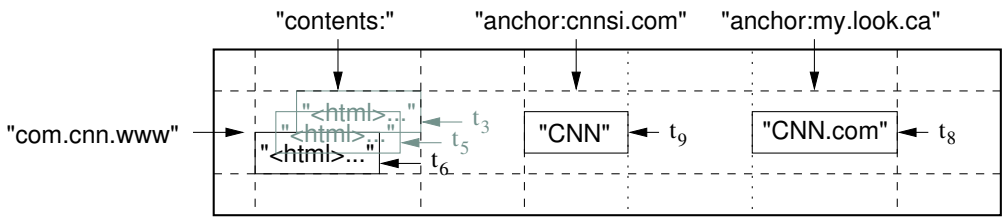


Figure 1: A slice of an example table that stores Web pages. The row name is a reversed URL. The contents column family contains the page contents, and the anchor column family contains the text of any anchors that reference the page. CNN's home page is referenced by both the Sports Illustrated and the MY-look home pages, so the row contains columns named anchor:cnnsi.com and anchor:my.look.ca. Each anchor cell has one version; the contents column has three versions, at timestamps t_3, t_5, and t_6.

We settled on this data model after examining a variety of potential uses of a Bigtable-like system. As one concrete example that drove some of our design decisions, suppose we want to keep a copy of a large collection of web pages and related information that could be used by many different projects; let us call this particular table the *Webtable*. In Webtable, we would use URLs as row keys, various aspects of web pages as column names, and store the contents of the web pages in the contents: column under the timestamps when they were fetched, as illustrated in Figure 1.

Rows

The row keys in a table are arbitrary strings (currently up to 64KB in size, although 10-100 bytes is a typical size for most of our users). Every read or write of data under a single row key is atomic (regardless of the number of different columns being read or written in the row), a design decision that makes it easier for clients to reason about the system's behavior in the presence of concurrent updates to the same row.

Bigtable maintains data in lexicographic order by row key. The row range for a table is dynamically partitioned. Each row range is called a *tablet*, which is the unit of distribution and load balancing. As a result, reads of short row ranges are efficient and typically require communication with only a small number of machines. Clients can exploit this property by selecting their row keys so that they get good locality for their data accesses. For example, in Webtable, pages in the same domain are grouped together into contiguous rows by reversing the hostname components of the URLs. For example, we store data for maps.google.com/index.html under the key com.google.maps/index.html. Storing pages from the same domain near each other makes some host and domain analyses more efficient.

Column Families

Column keys are grouped into sets called *column families*, which form the basic unit of access control. All data stored in a column family is usually of the same type (we compress data in the same column family together). A column family must be created before data can be stored under any column key in that family; after a family has been created, any column key within the family can be used. It is our intent that the number of distinct column families in a table be small (in the hundreds at most), and that families rarely change during operation. In contrast, a table may have an unbounded number of columns.

A column key is named using the following syntax: *family*:*qualifier*. Column family names must be printable, but qualifiers may be arbitrary strings. An example column family for the Webtable is language, which stores the language in which a web page was written. We use only one column key in the language family, and it stores each web page's language ID. Another useful column family for this table is anchor; each column key in this family represents a single anchor, as shown in Figure 1. The qualifier is the name of the referring site; the cell contents is the link text.

Access control and both disk and memory accounting are performed at the column-family level. In our Webtable example, these controls allow us to manage several different types of applications: some that add new base data, some that read the base data and create derived column families, and some that are only allowed to view existing data (and possibly not even to view all of the existing families for privacy reasons).

Timestamps

Each cell in a Bigtable can contain multiple versions of the same data; these versions are indexed by timestamp. Bigtable timestamps are 64-bit integers. They can be assigned by Bigtable, in which case they represent "real time" in microseconds, or be explicitly assigned by client

```
// Open the table
Table *T = OpenOrDie("/bigtable/web/webtable");

// Write a new anchor and delete an old anchor
RowMutation r1(T, "com.cnn.www");
r1.Set("anchor:www.c-span.org", "CNN");
r1.Delete("anchor:www.abc.com");
Operation op;
Apply(&op, &r1);
```

Figure 2: Writing to Bigtable.

```
Scanner scanner(T);
ScanStream *stream;
stream = scanner.FetchColumnFamily("anchor");
stream->SetReturnAllVersions();
scanner.Lookup("com.cnn.www");
for (; !stream->Done(); stream->Next()) {
  printf("%s %s %lld %s\n",
         scanner.RowName(),
         stream->ColumnName(),
         stream->MicroTimestamp(),
         stream->Value());
}
```

Figure 3: Reading from Bigtable.

applications. Applications that need to avoid collisions must generate unique timestamps themselves. Different versions of a cell are stored in decreasing timestamp order, so that the most recent versions can be read first.

To make the management of versioned data less onerous, we support two per-column-family settings that tell Bigtable to garbage-collect cell versions automatically. The client can specify either that only the last n versions of a cell be kept, or that only new-enough versions be kept (e.g., only keep values that were written in the last seven days).

In our Webtable example, we set the timestamps of the crawled pages stored in the contents: column to the times at which these page versions were actually crawled. The garbage-collection mechanism described above lets us keep only the most recent three versions of every page.

3 API

The Bigtable API provides functions for creating and deleting tables and column families. It also provides functions for changing cluster, table, and column family metadata, such as access control rights.

Client applications can write or delete values in Bigtable, look up values from individual rows, or iterate over a subset of the data in a table. Figure 2 shows C++ code that uses a `RowMutation` abstraction to perform a series of updates. (Irrelevant details were elided to keep the example short.) The call to `Apply` performs an atomic mutation to the Webtable: it adds one anchor to www.cnn.com and deletes a different anchor.

Figure 3 shows C++ code that uses a `Scanner` abstraction to iterate over all anchors in a particular row. Clients can iterate over multiple column families, and there are several mechanisms for limiting the rows, columns, and timestamps produced by a scan. For example, we could restrict the scan above to only produce anchors whose columns match the regular expression anchor:*.cnn.com, or to only produce anchors whose timestamps fall within ten days of the current time.

Bigtable supports several other features that allow the user to manipulate data in more complex ways. First, Bigtable supports single-row transactions, which can be used to perform atomic read-modify-write sequences on data stored under a single row key. Bigtable does not currently support general transactions across row keys, although it provides an interface for batching writes across row keys at the clients. Second, Bigtable allows cells to be used as integer counters. Finally, Bigtable supports the execution of client-supplied scripts in the address spaces of the servers. The scripts are written in a language developed at Google for processing data called Sawzall [28]. At the moment, our Sawzall-based API does not allow client scripts to write back into Bigtable, but it does allow various forms of data transformation, filtering based on arbitrary expressions, and summarization via a variety of operators.

Bigtable can be used with MapReduce [12], a framework for running large-scale parallel computations developed at Google. We have written a set of wrappers that allow a Bigtable to be used both as an input source and as an output target for MapReduce jobs.

4 Building Blocks

Bigtable is built on several other pieces of Google infrastructure. Bigtable uses the distributed Google File System (GFS) [17] to store log and data files. A Bigtable cluster typically operates in a shared pool of machines that run a wide variety of other distributed applications, and Bigtable processes often share the same machines with processes from other applications. Bigtable depends on a cluster management system for scheduling jobs, managing resources on shared machines, dealing with machine failures, and monitoring machine status.

The Google *SSTable* file format is used internally to store Bigtable data. An SSTable provides a persistent, ordered immutable map from keys to values, where both keys and values are arbitrary byte strings. Operations are provided to look up the value associated with a specified

key, and to iterate over all key/value pairs in a specified key range. Internally, each SSTable contains a sequence of blocks (typically each block is 64KB in size, but this is configurable). A block index (stored at the end of the SSTable) is used to locate blocks; the index is loaded into memory when the SSTable is opened. A lookup can be performed with a single disk seek: we first find the appropriate block by performing a binary search in the in-memory index, and then reading the appropriate block from disk. Optionally, an SSTable can be completely mapped into memory, which allows us to perform lookups and scans without touching disk.

Bigtable relies on a highly-available and persistent distributed lock service called Chubby [8]. A Chubby service consists of five active replicas, one of which is elected to be the master and actively serve requests. The service is live when a majority of the replicas are running and can communicate with each other. Chubby uses the Paxos algorithm [9, 23] to keep its replicas consistent in the face of failure. Chubby provides a namespace that consists of directories and small files. Each directory or file can be used as a lock, and reads and writes to a file are atomic. The Chubby client library provides consistent caching of Chubby files. Each Chubby client maintains a *session* with a Chubby service. A client's session expires if it is unable to renew its session lease within the lease expiration time. When a client's session expires, it loses any locks and open handles. Chubby clients can also register callbacks on Chubby files and directories for notification of changes or session expiration.

Bigtable uses Chubby for a variety of tasks: to ensure that there is at most one active master at any time; to store the bootstrap location of Bigtable data (see Section 5.1); to discover tablet servers and finalize tablet server deaths (see Section 5.2); to store Bigtable schema information (the column family information for each table); and to store access control lists. If Chubby becomes unavailable for an extended period of time, Bigtable becomes unavailable. We recently measured this effect in 14 Bigtable clusters spanning 11 Chubby instances. The average percentage of Bigtable server hours during which some data stored in Bigtable was not available due to Chubby unavailability (caused by either Chubby outages or network issues) was 0.0047%. The percentage for the single cluster that was most affected by Chubby unavailability was 0.0326%.

5 Implementation

The Bigtable implementation has three major components: a library that is linked into every client, one master server, and many tablet servers. Tablet servers can be dynamically added (or removed) from a cluster to accomodate changes in workloads.

The master is responsible for assigning tablets to tablet servers, detecting the addition and expiration of tablet servers, balancing tablet-server load, and garbage collection of files in GFS. In addition, it handles schema changes such as table and column family creations.

Each tablet server manages a set of tablets (typically we have somewhere between ten to a thousand tablets per tablet server). The tablet server handles read and write requests to the tablets that it has loaded, and also splits tablets that have grown too large.

As with many single-master distributed storage systems [17, 21], client data does not move through the master: clients communicate directly with tablet servers for reads and writes. Because Bigtable clients do not rely on the master for tablet location information, most clients never communicate with the master. As a result, the master is lightly loaded in practice.

A Bigtable cluster stores a number of tables. Each table consists of a set of tablets, and each tablet contains all data associated with a row range. Initially, each table consists of just one tablet. As a table grows, it is automatically split into multiple tablets, each approximately 100-200 MB in size by default.

5.1 Tablet Location

We use a three-level hierarchy analogous to that of a B^+-tree [10] to store tablet location information (Figure 4).

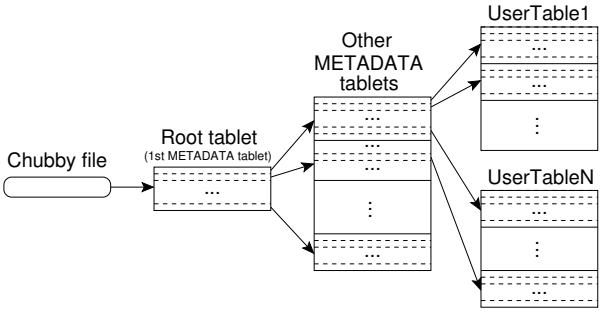


Figure 4: Tablet location hierarchy.

The first level is a file stored in Chubby that contains the location of the *root tablet*. The *root tablet* contains the location of all tablets in a special `METADATA` table. Each `METADATA` tablet contains the location of a set of user tablets. The *root tablet* is just the first tablet in the `METADATA` table, but is treated specially—it is never split—to ensure that the tablet location hierarchy has no more than three levels.

The `METADATA` table stores the location of a tablet under a row key that is an encoding of the tablet's table

identifier and its end row. Each METADATA row stores approximately 1KB of data in memory. With a modest limit of 128 MB METADATA tablets, our three-level location scheme is sufficient to address 2^{34} tablets (or 2^{61} bytes in 128 MB tablets).

The client library caches tablet locations. If the client does not know the location of a tablet, or if it discovers that cached location information is incorrect, then it recursively moves up the tablet location hierarchy. If the client's cache is empty, the location algorithm requires three network round-trips, including one read from Chubby. If the client's cache is stale, the location algorithm could take up to six round-trips, because stale cache entries are only discovered upon misses (assuming that METADATA tablets do not move very frequently). Although tablet locations are stored in memory, so no GFS accesses are required, we further reduce this cost in the common case by having the client library prefetch tablet locations: it reads the metadata for more than one tablet whenever it reads the METADATA table.

We also store secondary information in the METADATA table, including a log of all events pertaining to each tablet (such as when a server begins serving it). This information is helpful for debugging and performance analysis.

5.2 Tablet Assignment

Each tablet is assigned to one tablet server at a time. The master keeps track of the set of live tablet servers, and the current assignment of tablets to tablet servers, including which tablets are unassigned. When a tablet is unassigned, and a tablet server with sufficient room for the tablet is available, the master assigns the tablet by sending a tablet load request to the tablet server.

Bigtable uses Chubby to keep track of tablet servers. When a tablet server starts, it creates, and acquires an exclusive lock on, a uniquely-named file in a specific Chubby directory. The master monitors this directory (the *servers directory*) to discover tablet servers. A tablet server stops serving its tablets if it loses its exclusive lock: e.g., due to a network partition that caused the server to lose its Chubby session. (Chubby provides an efficient mechanism that allows a tablet server to check whether it still holds its lock without incurring network traffic.) A tablet server will attempt to reacquire an exclusive lock on its file as long as the file still exists. If the file no longer exists, then the tablet server will never be able to serve again, so it kills itself. Whenever a tablet server terminates (e.g., because the cluster management system is removing the tablet server's machine from the cluster), it attempts to release its lock so that the master will reassign its tablets more quickly.

The master is responsible for detecting when a tablet server is no longer serving its tablets, and for reassigning those tablets as soon as possible. To detect when a tablet server is no longer serving its tablets, the master periodically asks each tablet server for the status of its lock. If a tablet server reports that it has lost its lock, or if the master was unable to reach a server during its last several attempts, the master attempts to acquire an exclusive lock on the server's file. If the master is able to acquire the lock, then Chubby is live and the tablet server is either dead or having trouble reaching Chubby, so the master ensures that the tablet server can never serve again by deleting its server file. Once a server's file has been deleted, the master can move all the tablets that were previously assigned to that server into the set of unassigned tablets. To ensure that a Bigtable cluster is not vulnerable to networking issues between the master and Chubby, the master kills itself if its Chubby session expires. However, as described above, master failures do not change the assignment of tablets to tablet servers.

When a master is started by the cluster management system, it needs to discover the current tablet assignments before it can change them. The master executes the following steps at startup. (1) The master grabs a unique *master* lock in Chubby, which prevents concurrent master instantiations. (2) The master scans the servers directory in Chubby to find the live servers. (3) The master communicates with every live tablet server to discover what tablets are already assigned to each server. (4) The master scans the METADATA table to learn the set of tablets. Whenever this scan encounters a tablet that is not already assigned, the master adds the tablet to the set of unassigned tablets, which makes the tablet eligible for tablet assignment.

One complication is that the scan of the METADATA table cannot happen until the METADATA tablets have been assigned. Therefore, before starting this scan (step 4), the master adds the root tablet to the set of unassigned tablets if an assignment for the root tablet was not discovered during step 3. This addition ensures that the root tablet will be assigned. Because the root tablet contains the names of all METADATA tablets, the master knows about all of them after it has scanned the root tablet.

The set of existing tablets only changes when a table is created or deleted, two existing tablets are merged to form one larger tablet, or an existing tablet is split into two smaller tablets. The master is able to keep track of these changes because it initiates all but the last. Tablet splits are treated specially since they are initiated by a tablet server. The tablet server commits the split by recording information for the new tablet in the METADATA table. When the split has committed, it notifies the master. In case the split notification is lost (either

because the tablet server or the master died), the master detects the new tablet when it asks a tablet server to load the tablet that has now split. The tablet server will notify the master of the split, because the tablet entry it finds in the METADATA table will specify only a portion of the tablet that the master asked it to load.

5.3 Tablet Serving

The persistent state of a tablet is stored in GFS, as illustrated in Figure 5. Updates are committed to a commit log that stores redo records. Of these updates, the recently committed ones are stored in memory in a sorted buffer called a *memtable*; the older updates are stored in a sequence of SSTables. To recover a tablet, a tablet server

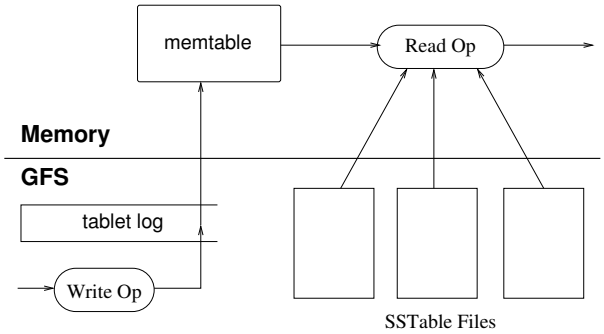


Figure 5: Tablet Representation

reads its metadata from the METADATA table. This metadata contains the list of SSTables that comprise a tablet and a set of a redo points, which are pointers into any commit logs that may contain data for the tablet. The server reads the indices of the SSTables into memory and reconstructs the memtable by applying all of the updates that have committed since the redo points.

When a write operation arrives at a tablet server, the server checks that it is well-formed, and that the sender is authorized to perform the mutation. Authorization is performed by reading the list of permitted writers from a Chubby file (which is almost always a hit in the Chubby client cache). A valid mutation is written to the commit log. Group commit is used to improve the throughput of lots of small mutations [13, 16]. After the write has been committed, its contents are inserted into the memtable.

When a read operation arrives at a tablet server, it is similarly checked for well-formedness and proper authorization. A valid read operation is executed on a merged view of the sequence of SSTables and the memtable. Since the SSTables and the memtable are lexicographically sorted data structures, the merged view can be formed efficiently.

Incoming read and write operations can continue while tablets are split and merged.

5.4 Compactions

As write operations execute, the size of the memtable increases. When the memtable size reaches a threshold, the memtable is frozen, a new memtable is created, and the frozen memtable is converted to an SSTable and written to GFS. This *minor compaction* process has two goals: it shrinks the memory usage of the tablet server, and it reduces the amount of data that has to be read from the commit log during recovery if this server dies. Incoming read and write operations can continue while compactions occur.

Every minor compaction creates a new SSTable. If this behavior continued unchecked, read operations might need to merge updates from an arbitrary number of SSTables. Instead, we bound the number of such files by periodically executing a *merging compaction* in the background. A merging compaction reads the contents of a few SSTables and the memtable, and writes out a new SSTable. The input SSTables and memtable can be discarded as soon as the compaction has finished.

A merging compaction that rewrites all SSTables into exactly one SSTable is called a *major compaction*. SSTables produced by non-major compactions can contain special deletion entries that suppress deleted data in older SSTables that are still live. A major compaction, on the other hand, produces an SSTable that contains no deletion information or deleted data. Bigtable cycles through all of its tablets and regularly applies major compactions to them. These major compactions allow Bigtable to reclaim resources used by deleted data, and also allow it to ensure that deleted data disappears from the system in a timely fashion, which is important for services that store sensitive data.

6 Refinements

The implementation described in the previous section required a number of refinements to achieve the high performance, availability, and reliability required by our users. This section describes portions of the implementation in more detail in order to highlight these refinements.

Locality groups

Clients can group multiple column families together into a *locality group*. A separate SSTable is generated for each locality group in each tablet. Segregating column families that are not typically accessed together into separate locality groups enables more efficient reads. For example, page metadata in Webtable (such as language and checksums) can be in one locality group, and the contents of the page can be in a different group: an ap-

plication that wants to read the metadata does not need to read through all of the page contents.

In addition, some useful tuning parameters can be specified on a per-locality group basis. For example, a locality group can be declared to be in-memory. SSTables for in-memory locality groups are loaded lazily into the memory of the tablet server. Once loaded, column families that belong to such locality groups can be read without accessing the disk. This feature is useful for small pieces of data that are accessed frequently: we use it internally for the location column family in the `METADATA` table.

Compression

Clients can control whether or not the SSTables for a locality group are compressed, and if so, which compression format is used. The user-specified compression format is applied to each SSTable block (whose size is controllable via a locality group specific tuning parameter). Although we lose some space by compressing each block separately, we benefit in that small portions of an SSTable can be read without decompressing the entire file. Many clients use a two-pass custom compression scheme. The first pass uses Bentley and McIlroy's scheme [6], which compresses long common strings across a large window. The second pass uses a fast compression algorithm that looks for repetitions in a small 16 KB window of the data. Both compression passes are very fast—they encode at 100–200 MB/s, and decode at 400–1000 MB/s on modern machines.

Even though we emphasized speed instead of space reduction when choosing our compression algorithms, this two-pass compression scheme does surprisingly well. For example, in Webtable, we use this compression scheme to store Web page contents. In one experiment, we stored a large number of documents in a compressed locality group. For the purposes of the experiment, we limited ourselves to one version of each document instead of storing all versions available to us. The scheme achieved a 10-to-1 reduction in space. This is much better than typical Gzip reductions of 3-to-1 or 4-to-1 on HTML pages because of the way Webtable rows are laid out: all pages from a single host are stored close to each other. This allows the Bentley-McIlroy algorithm to identify large amounts of shared boilerplate in pages from the same host. Many applications, not just Webtable, choose their row names so that similar data ends up clustered, and therefore achieve very good compression ratios. Compression ratios get even better when we store multiple versions of the same value in Bigtable.

Caching for read performance

To improve read performance, tablet servers use two levels of caching. The Scan Cache is a higher-level cache that caches the key-value pairs returned by the SSTable interface to the tablet server code. The Block Cache is a lower-level cache that caches SSTables blocks that were read from GFS. The Scan Cache is most useful for applications that tend to read the same data repeatedly. The Block Cache is useful for applications that tend to read data that is close to the data they recently read (e.g., sequential reads, or random reads of different columns in the same locality group within a hot row).

Bloom filters

As described in Section 5.3, a read operation has to read from all SSTables that make up the state of a tablet. If these SSTables are not in memory, we may end up doing many disk accesses. We reduce the number of accesses by allowing clients to specify that Bloom filters [7] should be created for SSTables in a particular locality group. A Bloom filter allows us to ask whether an SSTable might contain any data for a specified row/column pair. For certain applications, a small amount of tablet server memory used for storing Bloom filters drastically reduces the number of disk seeks required for read operations. Our use of Bloom filters also implies that most lookups for non-existent rows or columns do not need to touch disk.

Commit-log implementation

If we kept the commit log for each tablet in a separate log file, a very large number of files would be written concurrently in GFS. Depending on the underlying file system implementation on each GFS server, these writes could cause a large number of disk seeks to write to the different physical log files. In addition, having separate log files per tablet also reduces the effectiveness of the group commit optimization, since groups would tend to be smaller. To fix these issues, we append mutations to a single commit log per tablet server, co-mingling mutations for different tablets in the same physical log file [18, 20].

Using one log provides significant performance benefits during normal operation, but it complicates recovery. When a tablet server dies, the tablets that it served will be moved to a large number of other tablet servers: each server typically loads a small number of the original server's tablets. To recover the state for a tablet, the new tablet server needs to reapply the mutations for that tablet from the commit log written by the original tablet server. However, the mutations for these tablets

were co-mingled in the same physical log file. One approach would be for each new tablet server to read this full commit log file and apply just the entries needed for the tablets it needs to recover. However, under such a scheme, if 100 machines were each assigned a single tablet from a failed tablet server, then the log file would be read 100 times (once by each server).

We avoid duplicating log reads by first sorting the commit log entries in order of the keys $\langle$table, row name, log sequence number$\rangle$. In the sorted output, all mutations for a particular tablet are contiguous and can therefore be read efficiently with one disk seek followed by a sequential read. To parallelize the sorting, we partition the log file into 64 MB segments, and sort each segment in parallel on different tablet servers. This sorting process is coordinated by the master and is initiated when a tablet server indicates that it needs to recover mutations from some commit log file.

Writing commit logs to GFS sometimes causes performance hiccups for a variety of reasons (e.g., a GFS server machine involved in the write crashes, or the network paths traversed to reach the particular set of three GFS servers is suffering network congestion, or is heavily loaded). To protect mutations from GFS latency spikes, each tablet server actually has two log writing threads, each writing to its own log file; only one of these two threads is actively in use at a time. If writes to the active log file are performing poorly, the log file writing is switched to the other thread, and mutations that are in the commit log queue are written by the newly active log writing thread. Log entries contain sequence numbers to allow the recovery process to elide duplicated entries resulting from this log switching process.

Speeding up tablet recovery

If the master moves a tablet from one tablet server to another, the source tablet server first does a minor compaction on that tablet. This compaction reduces recovery time by reducing the amount of uncompacted state in the tablet server's commit log. After finishing this compaction, the tablet server stops serving the tablet. Before it actually unloads the tablet, the tablet server does another (usually very fast) minor compaction to eliminate any remaining uncompacted state in the tablet server's log that arrived while the first minor compaction was being performed. After this second minor compaction is complete, the tablet can be loaded on another tablet server without requiring any recovery of log entries.

Exploiting immutability

Besides the SSTable caches, various other parts of the Bigtable system have been simplified by the fact that all of the SSTables that we generate are immutable. For example, we do not need any synchronization of accesses to the file system when reading from SSTables. As a result, concurrency control over rows can be implemented very efficiently. The only mutable data structure that is accessed by both reads and writes is the memtable. To reduce contention during reads of the memtable, we make each memtable row copy-on-write and allow reads and writes to proceed in parallel.

Since SSTables are immutable, the problem of permanently removing deleted data is transformed to garbage collecting obsolete SSTables. Each tablet's SSTables are registered in the `METADATA` table. The master removes obsolete SSTables as a mark-and-sweep garbage collection [25] over the set of SSTables, where the `METADATA` table contains the set of roots.

Finally, the immutability of SSTables enables us to split tablets quickly. Instead of generating a new set of SSTables for each child tablet, we let the child tablets share the SSTables of the parent tablet.

7 Performance Evaluation

We set up a Bigtable cluster with N tablet servers to measure the performance and scalability of Bigtable as N is varied. The tablet servers were configured to use 1 GB of memory and to write to a GFS cell consisting of 1786 machines with two 400 GB IDE hard drives each. N client machines generated the Bigtable load used for these tests. (We used the same number of clients as tablet servers to ensure that clients were never a bottleneck.) Each machine had two dual-core Opteron 2 GHz chips, enough physical memory to hold the working set of all running processes, and a single gigabit Ethernet link. The machines were arranged in a two-level tree-shaped switched network with approximately 100-200 Gbps of aggregate bandwidth available at the root. All of the machines were in the same hosting facility and therefore the round-trip time between any pair of machines was less than a millisecond.

The tablet servers and master, test clients, and GFS servers all ran on the same set of machines. Every machine ran a GFS server. Some of the machines also ran either a tablet server, or a client process, or processes from other jobs that were using the pool at the same time as these experiments.

R is the distinct number of Bigtable row keys involved in the test. R was chosen so that each benchmark read or wrote approximately 1 GB of data per tablet server.

The *sequential write* benchmark used row keys with names 0 to $R - 1$. This space of row keys was partitioned into $10N$ equal-sized ranges. These ranges were assigned to the N clients by a central scheduler that as-

Experiment	# of Tablet Servers			
	1	50	250	500
random reads	1212	593	479	241
random reads (mem)	10811	8511	8000	6250
random writes	8850	3745	3425	2000
sequential reads	4425	2463	2625	2469
sequential writes	8547	3623	2451	1905
scans	15385	10526	9524	7843

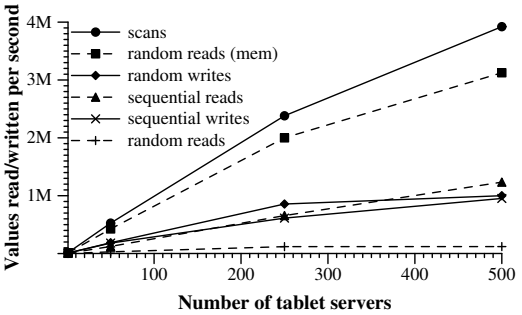


Figure 6: Number of 1000-byte values read/written per second. The table shows the rate per tablet server; the graph shows the aggregate rate.

signed the next available range to a client as soon as the client finished processing the previous range assigned to it. This dynamic assignment helped mitigate the effects of performance variations caused by other processes running on the client machines. We wrote a single string under each row key. Each string was generated randomly and was therefore uncompressible. In addition, strings under different row key were distinct, so no cross-row compression was possible. The *random write* benchmark was similar except that the row key was hashed modulo R immediately before writing so that the write load was spread roughly uniformly across the entire row space for the entire duration of the benchmark.

The *sequential read* benchmark generated row keys in exactly the same way as the sequential write benchmark, but instead of writing under the row key, it read the string stored under the row key (which was written by an earlier invocation of the sequential write benchmark). Similarly, the *random read* benchmark shadowed the operation of the random write benchmark.

The *scan* benchmark is similar to the sequential read benchmark, but uses support provided by the Bigtable API for scanning over all values in a row range. Using a scan reduces the number of RPCs executed by the benchmark since a single RPC fetches a large sequence of values from a tablet server.

The *random reads (mem)* benchmark is similar to the random read benchmark, but the locality group that contains the benchmark data is marked as *in-memory*, and therefore the reads are satisfied from the tablet server's memory instead of requiring a GFS read. For just this benchmark, we reduced the amount of data per tablet server from 1 GB to 100 MB so that it would fit comfortably in the memory available to the tablet server.

Figure 6 shows two views on the performance of our benchmarks when reading and writing 1000-byte values to Bigtable. The table shows the number of operations per second per tablet server; the graph shows the aggregate number of operations per second.

Single tablet-server performance

Let us first consider performance with just one tablet server. Random reads are slower than all other operations by an order of magnitude or more. Each random read involves the transfer of a 64 KB SSTable block over the network from GFS to a tablet server, out of which only a single 1000-byte value is used. The tablet server executes approximately 1200 reads per second, which translates into approximately 75 MB/s of data read from GFS. This bandwidth is enough to saturate the tablet server CPUs because of overheads in our networking stack, SSTable parsing, and Bigtable code, and is also almost enough to saturate the network links used in our system. Most Bigtable applications with this type of an access pattern reduce the block size to a smaller value, typically 8KB.

Random reads from memory are much faster since each 1000-byte read is satisfied from the tablet server's local memory without fetching a large 64 KB block from GFS.

Random and sequential writes perform better than random reads since each tablet server appends all incoming writes to a single commit log and uses group commit to stream these writes efficiently to GFS. There is no significant difference between the performance of random writes and sequential writes; in both cases, all writes to the tablet server are recorded in the same commit log.

Sequential reads perform better than random reads since every 64 KB SSTable block that is fetched from GFS is stored into our block cache, where it is used to serve the next 64 read requests.

Scans are even faster since the tablet server can return a large number of values in response to a single client RPC, and therefore RPC overhead is amortized over a large number of values.

Scaling

Aggregate throughput increases dramatically, by over a factor of a hundred, as we increase the number of tablet servers in the system from 1 to 500. For example, the

# of tablet servers	# of clusters
0 .. 19	259
20 .. 49	47
50 .. 99	20
100 .. 499	50
> 500	12

Table 1: Distribution of number of tablet servers in Bigtable clusters.

performance of random reads from memory increases by almost a factor of 300 as the number of tablet server increases by a factor of 500. This behavior occurs because the bottleneck on performance for this benchmark is the individual tablet server CPU.

However, performance does not increase linearly. For most benchmarks, there is a significant drop in per-server throughput when going from 1 to 50 tablet servers. This drop is caused by imbalance in load in multiple server configurations, often due to other processes contending for CPU and network. Our load balancing algorithm attempts to deal with this imbalance, but cannot do a perfect job for two main reasons: rebalancing is throttled to reduce the number of tablet movements (a tablet is unavailable for a short time, typically less than one second, when it is moved), and the load generated by our benchmarks shifts around as the benchmark progresses.

The random read benchmark shows the worst scaling (an increase in aggregate throughput by only a factor of 100 for a 500-fold increase in number of servers). This behavior occurs because (as explained above) we transfer one large 64KB block over the network for every 1000-byte read. This transfer saturates various shared 1 Gigabit links in our network and as a result, the per-server throughput drops significantly as we increase the number of machines.

8 Real Applications

As of August 2006, there are 388 non-test Bigtable clusters running in various Google machine clusters, with a combined total of about 24,500 tablet servers. Table 1 shows a rough distribution of tablet servers per cluster. Many of these clusters are used for development purposes and therefore are idle for significant periods. One group of 14 busy clusters with 8069 total tablet servers saw an aggregate volume of more than 1.2 million requests per second, with incoming RPC traffic of about 741 MB/s and outgoing RPC traffic of about 16 GB/s.

Table 2 provides some data about a few of the tables currently in use. Some tables store data that is served to users, whereas others store data for batch processing; the tables range widely in total size, average cell size, percentage of data served from memory, and complexity of the table schema. In the rest of this section, we briefly describe how three product teams use Bigtable.

8.1 Google Analytics

Google Analytics (analytics.google.com) is a service that helps webmasters analyze traffic patterns at their web sites. It provides aggregate statistics, such as the number of unique visitors per day and the page views per URL per day, as well as site-tracking reports, such as the percentage of users that made a purchase, given that they earlier viewed a specific page.

To enable the service, webmasters embed a small JavaScript program in their web pages. This program is invoked whenever a page is visited. It records various information about the request in Google Analytics, such as a user identifier and information about the page being fetched. Google Analytics summarizes this data and makes it available to webmasters.

We briefly describe two of the tables used by Google Analytics. The raw click table (~200 TB) maintains a row for each end-user session. The row name is a tuple containing the website's name and the time at which the session was created. This schema ensures that sessions that visit the same web site are contiguous, and that they are sorted chronologically. This table compresses to 14% of its original size.

The summary table (~20 TB) contains various predefined summaries for each website. This table is generated from the raw click table by periodically scheduled MapReduce jobs. Each MapReduce job extracts recent session data from the raw click table. The overall system's throughput is limited by the throughput of GFS. This table compresses to 29% of its original size.

8.2 Google Earth

Google operates a collection of services that provide users with access to high-resolution satellite imagery of the world's surface, both through the web-based Google Maps interface (maps.google.com) and through the Google Earth (earth.google.com) custom client software. These products allow users to navigate across the world's surface: they can pan, view, and annotate satellite imagery at many different levels of resolution. This system uses one table to preprocess data, and a different set of tables for serving client data.

The preprocessing pipeline uses one table to store raw imagery. During preprocessing, the imagery is cleaned and consolidated into final serving data. This table contains approximately 70 terabytes of data and therefore is served from disk. The images are efficiently compressed already, so Bigtable compression is disabled.

Project name	Table size (TB)	Compression ratio	# Cells (billions)	# Column Families	# Locality Groups	% in memory	Latency-sensitive?
Crawl	800	11%	1000	16	8	0%	No
Crawl	50	33%	200	2	2	0%	No
Google Analytics	20	29%	10	1	1	0%	Yes
Google Analytics	200	14%	80	1	1	0%	Yes
Google Base	2	31%	10	29	3	15%	Yes
Google Earth	0.5	64%	8	7	2	33%	Yes
Google Earth	70	–	9	8	3	0%	No
Orkut	9	–	0.9	8	5	1%	Yes
Personalized Search	4	47%	6	93	11	5%	Yes

Table 2: Characteristics of a few tables in production use. *Table size* (measured before compression) and *# Cells* indicate approximate sizes. *Compression ratio* is not given for tables that have compression disabled.

Each row in the imagery table corresponds to a single geographic segment. Rows are named to ensure that adjacent geographic segments are stored near each other. The table contains a column family to keep track of the sources of data for each segment. This column family has a large number of columns: essentially one for each raw data image. Since each segment is only built from a few images, this column family is very sparse.

The preprocessing pipeline relies heavily on MapReduce over Bigtable to transform data. The overall system processes over 1 MB/sec of data per tablet server during some of these MapReduce jobs.

The serving system uses one table to index data stored in GFS. This table is relatively small (~500 GB), but it must serve tens of thousands of queries per second per datacenter with low latency. As a result, this table is hosted across hundreds of tablet servers and contains in-memory column families.

8.3 Personalized Search

Personalized Search (www.google.com/psearch) is an opt-in service that records user queries and clicks across a variety of Google properties such as web search, images, and news. Users can browse their search histories to revisit their old queries and clicks, and they can ask for personalized search results based on their historical Google usage patterns.

Personalized Search stores each user's data in Bigtable. Each user has a unique userid and is assigned a row named by that userid. All user actions are stored in a table. A separate column family is reserved for each type of action (for example, there is a column family that stores all web queries). Each data element uses as its Bigtable timestamp the time at which the corresponding user action occurred. Personalized Search generates user profiles using a MapReduce over Bigtable. These user profiles are used to personalize live search results.

The Personalized Search data is replicated across several Bigtable clusters to increase availability and to reduce latency due to distance from clients. The Personalized Search team originally built a client-side replication mechanism on top of Bigtable that ensured eventual consistency of all replicas. The current system now uses a replication subsystem that is built into the servers.

The design of the Personalized Search storage system allows other groups to add new per-user information in their own columns, and the system is now used by many other Google properties that need to store per-user configuration options and settings. Sharing a table amongst many groups resulted in an unusually large number of column families. To help support sharing, we added a simple quota mechanism to Bigtable to limit the storage consumption by any particular client in shared tables; this mechanism provides some isolation between the various product groups using this system for per-user information storage.

9 Lessons

In the process of designing, implementing, maintaining, and supporting Bigtable, we gained useful experience and learned several interesting lessons.

One lesson we learned is that large distributed systems are vulnerable to many types of failures, not just the standard network partitions and fail-stop failures assumed in many distributed protocols. For example, we have seen problems due to all of the following causes: memory and network corruption, large clock skew, hung machines, extended and asymmetric network partitions, bugs in other systems that we are using (Chubby for example), overflow of GFS quotas, and planned and unplanned hardware maintenance. As we have gained more experience with these problems, we have addressed them by changing various protocols. For example, we added checksumming to our RPC mechanism. We also handled

some problems by removing assumptions made by one part of the system about another part. For example, we stopped assuming a given Chubby operation could return only one of a fixed set of errors.

Another lesson we learned is that it is important to delay adding new features until it is clear how the new features will be used. For example, we initially planned to support general-purpose transactions in our API. Because we did not have an immediate use for them, however, we did not implement them. Now that we have many real applications running on Bigtable, we have been able to examine their actual needs, and have discovered that most applications require only single-row transactions. Where people have requested distributed transactions, the most important use is for maintaining secondary indices, and we plan to add a specialized mechanism to satisfy this need. The new mechanism will be less general than distributed transactions, but will be more efficient (especially for updates that span hundreds of rows or more) and will also interact better with our scheme for optimistic cross-data-center replication.

A practical lesson that we learned from supporting Bigtable is the importance of proper system-level monitoring (i.e., monitoring both Bigtable itself, as well as the client processes using Bigtable). For example, we extended our RPC system so that for a sample of the RPCs, it keeps a detailed trace of the important actions done on behalf of that RPC. This feature has allowed us to detect and fix many problems such as lock contention on tablet data structures, slow writes to GFS while committing Bigtable mutations, and stuck accesses to the `METADATA` table when `METADATA` tablets are unavailable. Another example of useful monitoring is that every Bigtable cluster is registered in Chubby. This allows us to track down all clusters, discover how big they are, see which versions of our software they are running, how much traffic they are receiving, and whether or not there are any problems such as unexpectedly large latencies.

The most important lesson we learned is the value of simple designs. Given both the size of our system (about 100,000 lines of non-test code), as well as the fact that code evolves over time in unexpected ways, we have found that code and design clarity are of immense help in code maintenance and debugging. One example of this is our tablet-server membership protocol. Our first protocol was simple: the master periodically issued leases to tablet servers, and tablet servers killed themselves if their lease expired. Unfortunately, this protocol reduced availability significantly in the presence of network problems, and was also sensitive to master recovery time. We redesigned the protocol several times until we had a protocol that performed well. However, the resulting protocol was too complex and depended on the behavior of Chubby features that were seldom exercised by other applications. We discovered that we were spending an inordinate amount of time debugging obscure corner cases, not only in Bigtable code, but also in Chubby code. Eventually, we scrapped this protocol and moved to a newer simpler protocol that depends solely on widely-used Chubby features.

10 Related Work

The Boxwood project [24] has components that overlap in some ways with Chubby, GFS, and Bigtable, since it provides for distributed agreement, locking, distributed chunk storage, and distributed B-tree storage. In each case where there is overlap, it appears that the Boxwood's component is targeted at a somewhat lower level than the corresponding Google service. The Boxwood project's goal is to provide infrastructure for building higher-level services such as file systems or databases, while the goal of Bigtable is to directly support client applications that wish to store data.

Many recent projects have tackled the problem of providing distributed storage or higher-level services over wide area networks, often at "Internet scale." This includes work on distributed hash tables that began with projects such as CAN [29], Chord [32], Tapestry [37], and Pastry [30]. These systems address concerns that do not arise for Bigtable, such as highly variable bandwidth, untrusted participants, or frequent reconfiguration; decentralized control and Byzantine fault tolerance are not Bigtable goals.

In terms of the distributed data storage model that one might provide to application developers, we believe the key-value pair model provided by distributed B-trees or distributed hash tables is too limiting. Key-value pairs are a useful building block, but they should not be the only building block one provides to developers. The model we chose is richer than simple key-value pairs, and supports sparse semi-structured data. Nonetheless, it is still simple enough that it lends itself to a very efficient flat-file representation, and it is transparent enough (via locality groups) to allow our users to tune important behaviors of the system.

Several database vendors have developed parallel databases that can store large volumes of data. Oracle's Real Application Cluster database [27] uses shared disks to store data (Bigtable uses GFS) and a distributed lock manager (Bigtable uses Chubby). IBM's DB2 Parallel Edition [4] is based on a shared-nothing [33] architecture similar to Bigtable. Each DB2 server is responsible for a subset of the rows in a table which it stores in a local relational database. Both products provide a complete relational model with transactions.

Bigtable locality groups realize similar compression and disk read performance benefits observed for other systems that organize data on disk using column-based rather than row-based storage, including C-Store [1, 34] and commercial products such as Sybase IQ [15, 36], SenSage [31], KDB+ [22], and the ColumnBM storage layer in MonetDB/X100 [38]. Another system that does vertical and horizontal data partioning into flat files and achieves good data compression ratios is AT&T's Daytona database [19]. Locality groups do not support CPU-cache-level optimizations, such as those described by Ailamaki [2].

The manner in which Bigtable uses memtables and SSTables to store updates to tablets is analogous to the way that the Log-Structured Merge Tree [26] stores updates to index data. In both systems, sorted data is buffered in memory before being written to disk, and reads must merge data from memory and disk.

C-Store and Bigtable share many characteristics: both systems use a shared-nothing architecture and have two different data structures, one for recent writes, and one for storing long-lived data, with a mechanism for moving data from one form to the other. The systems differ significantly in their API: C-Store behaves like a relational database, whereas Bigtable provides a lower level read and write interface and is designed to support many thousands of such operations per second per server. C-Store is also a "read-optimized relational DBMS", whereas Bigtable provides good performance on both read-intensive and write-intensive applications.

Bigtable's load balancer has to solve some of the same kinds of load and memory balancing problems faced by shared-nothing databases (e.g., [11, 35]). Our problem is somewhat simpler: (1) we do not consider the possibility of multiple copies of the same data, possibly in alternate forms due to views or indices; (2) we let the user tell us what data belongs in memory and what data should stay on disk, rather than trying to determine this dynamically; (3) we have no complex queries to execute or optimize.

11 Conclusions

We have described Bigtable, a distributed system for storing structured data at Google. Bigtable clusters have been in production use since April 2005, and we spent roughly seven person-years on design and implementation before that date. As of August 2006, more than sixty projects are using Bigtable. Our users like the performance and high availability provided by the Bigtable implementation, and that they can scale the capacity of their clusters by simply adding more machines to the system as their resource demands change over time.

Given the unusual interface to Bigtable, an interesting question is how difficult it has been for our users to adapt to using it. New users are sometimes uncertain of how to best use the Bigtable interface, particularly if they are accustomed to using relational databases that support general-purpose transactions. Nevertheless, the fact that many Google products successfully use Bigtable demonstrates that our design works well in practice.

We are in the process of implementing several additional Bigtable features, such as support for secondary indices and infrastructure for building cross-data-center replicated Bigtables with multiple master replicas. We have also begun deploying Bigtable as a service to product groups, so that individual groups do not need to maintain their own clusters. As our service clusters scale, we will need to deal with more resource-sharing issues within Bigtable itself [3, 5].

Finally, we have found that there are significant advantages to building our own storage solution at Google. We have gotten a substantial amount of flexibility from designing our own data model for Bigtable. In addition, our control over Bigtable's implementation, and the other Google infrastructure upon which Bigtable depends, means that we can remove bottlenecks and inefficiencies as they arise.

Acknowledgements

We thank the anonymous reviewers, David Nagle, and our shepherd Brad Calder, for their feedback on this paper. The Bigtable system has benefited greatly from the feedback of our many users within Google. In addition, we thank the following people for their contributions to Bigtable: Dan Aguayo, Sameer Ajmani, Zhifeng Chen, Bill Coughran, Mike Epstein, Healfdene Goguen, Robert Griesemer, Jeremy Hylton, Josh Hyman, Alex Khesin, Joanna Kulik, Alberto Lerner, Sherry Listgarten, Mike Maloney, Eduardo Pinheiro, Kathy Polizzi, Frank Yellin, and Arthur Zwiegincew.

References

[1] Abadi, D. J., Madden, S. R., and Ferreira, M. C. Integrating compression and execution in column-oriented database systems. *Proc. of SIGMOD* (2006).

[2] Ailamaki, A., DeWitt, D. J., Hill, M. D., and Skounakis, M. Weaving relations for cache performance. In *The VLDB Journal* (2001), pp. 169–180.

[3] Banga, G., Druschel, P., and Mogul, J. C. Resource containers: A new facility for resource management in server systems. In *Proc. of the 3rd OSDI* (Feb. 1999), pp. 45–58.

[4] Baru, C. K., Fecteau, G., Goyal, A., Hsiao, H., Jhingran, A., Padmanabhan, S., Copeland,

G. P., AND WILSON, W. G. DB2 parallel edition. *IBM Systems Journal 34*, 2 (1995), 292–322.

[5] BAVIER, A., BOWMAN, M., CHUN, B., CULLER, D., KARLIN, S., PETERSON, L., ROSCOE, T., SPALINK, T., AND WAWRZONIAK, M. Operating system support for planetary-scale network services. In *Proc. of the 1st NSDI* (Mar. 2004), pp. 253–266.

[6] BENTLEY, J. L., AND MCILROY, M. D. Data compression using long common strings. In *Data Compression Conference* (1999), pp. 287–295.

[7] BLOOM, B. H. Space/time trade-offs in hash coding with allowable errors. *CACM 13*, 7 (1970), 422–426.

[8] BURROWS, M. The Chubby lock service for loosely-coupled distributed systems. In *Proc. of the 7th OSDI* (Nov. 2006).

[9] CHANDRA, T., GRIESEMER, R., AND REDSTONE, J. Paxos made live — An engineering perspective. In *Proc. of PODC* (2007).

[10] COMER, D. Ubiquitous B-tree. *Computing Surveys 11*, 2 (June 1979), 121–137.

[11] COPELAND, G. P., ALEXANDER, W., BOUGHTER, E. E., AND KELLER, T. W. Data placement in Bubba. In *Proc. of SIGMOD* (1988), pp. 99–108.

[12] DEAN, J., AND GHEMAWAT, S. MapReduce: Simplified data processing on large clusters. In *Proc. of the 6th OSDI* (Dec. 2004), pp. 137–150.

[13] DEWITT, D., KATZ, R., OLKEN, F., SHAPIRO, L., STONEBRAKER, M., AND WOOD, D. Implementation techniques for main memory database systems. In *Proc. of SIGMOD* (June 1984), pp. 1–8.

[14] DEWITT, D. J., AND GRAY, J. Parallel database systems: The future of high performance database systems. *CACM 35*, 6 (June 1992), 85–98.

[15] FRENCH, C. D. One size fits all database architectures do not work for DSS. In *Proc. of SIGMOD* (May 1995), pp. 449–450.

[16] GAWLICK, D., AND KINKADE, D. Varieties of concurrency control in IMS/VS fast path. *Database Engineering Bulletin 8*, 2 (1985), 3–10.

[17] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The Google file system. In *Proc. of the 19th ACM SOSP* (Dec. 2003), pp. 29–43.

[18] GRAY, J. Notes on database operating systems. In *Operating Systems — An Advanced Course*, vol. 60 of *Lecture Notes in Computer Science*. Springer-Verlag, 1978.

[19] GREER, R. Daytona and the fourth-generation language Cymbal. In *Proc. of SIGMOD* (1999), pp. 525–526.

[20] HAGMANN, R. Reimplementing the Cedar file system using logging and group commit. In *Proc. of the 11th SOSP* (Dec. 1987), pp. 155–162.

[21] HARTMAN, J. H., AND OUSTERHOUT, J. K. The Zebra striped network file system. In *Proc. of the 14th SOSP* (Asheville, NC, 1993), pp. 29–43.

[22] KX.COM. kx.com/products/database.php. Product page.

[23] LAMPORT, L. The part-time parliament. *ACM TOCS 16*, 2 (1998), 133–169.

[24] MACCORMICK, J., MURPHY, N., NAJORK, M., THEKKATH, C. A., AND ZHOU, L. Boxwood: Abstractions as the foundation for storage infrastructure. In *Proc. of the 6th OSDI* (Dec. 2004), pp. 105–120.

[25] MCCARTHY, J. Recursive functions of symbolic expressions and their computation by machine. *CACM 3*, 4 (Apr. 1960), 184–195.

[26] O'NEIL, P., CHENG, E., GAWLICK, D., AND O'NEIL, E. The log-structured merge-tree (LSM-tree). *Acta Inf. 33*, 4 (1996), 351–385.

[27] ORACLE.COM. www.oracle.com/technology/products/-database/clustering/index.html. Product page.

[28] PIKE, R., DORWARD, S., GRIESEMER, R., AND QUINLAN, S. Interpreting the data: Parallel analysis with Sawzall. *Scientific Programming Journal 13*, 4 (2005), 227–298.

[29] RATNASAMY, S., FRANCIS, P., HANDLEY, M., KARP, R., AND SHENKER, S. A scalable content-addressable network. In *Proc. of SIGCOMM* (Aug. 2001), pp. 161–172.

[30] ROWSTRON, A., AND DRUSCHEL, P. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proc. of Middleware 2001* (Nov. 2001), pp. 329–350.

[31] SENSAGE.COM. sensage.com/products-sensage.htm. Product page.

[32] STOICA, I., MORRIS, R., KARGER, D., KAASHOEK, M. F., AND BALAKRISHNAN, H. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proc. of SIGCOMM* (Aug. 2001), pp. 149–160.

[33] STONEBRAKER, M. The case for shared nothing. *Database Engineering Bulletin 9*, 1 (Mar. 1986), 4–9.

[34] STONEBRAKER, M., ABADI, D. J., BATKIN, A., CHEN, X., CHERNIACK, M., FERREIRA, M., LAU, E., LIN, A., MADDEN, S., O'NEIL, E., O'NEIL, P., RASIN, A., TRAN, N., AND ZDONIK, S. C-Store: A column-oriented DBMS. In *Proc. of VLDB* (Aug. 2005), pp. 553–564.

[35] STONEBRAKER, M., AOKI, P. M., DEVINE, R., LITWIN, W., AND OLSON, M. A. Mariposa: A new architecture for distributed data. In *Proc. of the Tenth ICDE* (1994), IEEE Computer Society, pp. 54–65.

[36] SYBASE.COM. www.sybase.com/products/database-servers/sybaseiq. Product page.

[37] ZHAO, B. Y., KUBIATOWICZ, J., AND JOSEPH, A. D. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Tech. Rep. UCB/CSD-01-1141, CS Division, UC Berkeley, Apr. 2001.

[38] ZUKOWSKI, M., BONCZ, P. A., NES, N., AND HEMAN, S. MonetDB/X100 — A DBMS in the CPU cache. *IEEE Data Eng. Bull. 28*, 2 (2005), 17–22.

Chapter 10

Dynamo: Amazon's Highly Available Key-value Store

Authors:
Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall and Werner Vogels

Publication year: 2007

Dynamo: Amazon's Highly Available Key-value Store

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall and Werner Vogels

Amazon.com

ABSTRACT

Reliability at massive scale is one of the biggest challenges we face at Amazon.com, one of the largest e-commerce operations in the world; even the slightest outage has significant financial consequences and impacts customer trust. The Amazon.com platform, which provides services for many web sites worldwide, is implemented on top of an infrastructure of tens of thousands of servers and network components located in many datacenters around the world. At this scale, small and large components fail continuously and the way persistent state is managed in the face of these failures drives the reliability and scalability of the software systems.

This paper presents the design and implementation of Dynamo, a highly available key-value storage system that some of Amazon's core services use to provide an "always-on" experience. To achieve this level of availability, Dynamo sacrifices consistency under certain failure scenarios. It makes extensive use of object versioning and application-assisted conflict resolution in a manner that provides a novel interface for developers to use.

Categories and Subject Descriptors

D.4.2 [**Operating Systems**]: Storage Management; D.4.5 [**Operating Systems**]: Reliability; D.4.2 [**Operating Systems**]: Performance;

General Terms

Algorithms, Management, Measurement, Performance, Design, Reliability.

1. INTRODUCTION

Amazon runs a world-wide e-commerce platform that serves tens of millions customers at peak times using tens of thousands of servers located in many data centers around the world. There are strict operational requirements on Amazon's platform in terms of performance, reliability and efficiency, and to support continuous growth the platform needs to be highly scalable. Reliability is one of the most important requirements because even the slightest outage has significant financial consequences and impacts customer trust. In addition, to support continuous growth, the platform needs to be highly scalable.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SOSP 2007, Oct., 2004, Stevenson, WA, USA.
Copyright 2007 ACM XXX…$5.00.

One of the lessons our organization has learned from operating Amazon's platform is that the reliability and scalability of a system is dependent on how its application state is managed. Amazon uses a highly decentralized, loosely coupled, service oriented architecture consisting of hundreds of services. In this environment there is a particular need for storage technologies that are always available. For example, customers should be able to view and add items to their shopping cart even if disks are failing, network routes are flapping, or data centers are being destroyed by tornados. Therefore, the service responsible for managing shopping carts requires that it can always write to and read from its data store, and that its data needs to be available across multiple data centers.

Dealing with failures in an infrastructure comprised of millions of components is our standard mode of operation; there are always a small but significant number of server and network components that are failing at any given time. As such Amazon's software systems need to be constructed in a manner that treats failure handling as the normal case without impacting availability or performance.

To meet the reliability and scaling needs, Amazon has developed a number of storage technologies, of which the Amazon Simple Storage Service (also available outside of Amazon and known as Amazon S3), is probably the best known. This paper presents the design and implementation of Dynamo, another highly available and scalable distributed data store built for Amazon's platform. Dynamo is used to manage the state of services that have very high reliability requirements and need tight control over the tradeoffs between availability, consistency, cost-effectiveness and performance. Amazon's platform has a very diverse set of applications with different storage requirements. A select set of applications requires a storage technology that is flexible enough to let application designers configure their data store appropriately based on these tradeoffs to achieve high availability and guaranteed performance in the most cost effective manner.

There are many services on Amazon's platform that only need primary-key access to a data store. For many services, such as those that provide best seller lists, shopping carts, customer preferences, session management, sales rank, and product catalog, the common pattern of using a relational database would lead to inefficiencies and limit scale and availability. Dynamo provides a simple primary-key only interface to meet the requirements of these applications.

Dynamo uses a synthesis of well known techniques to achieve scalability and availability: Data is partitioned and replicated using consistent hashing [10], and consistency is facilitated by object versioning [12]. The consistency among replicas during updates is maintained by a quorum-like technique and a decentralized replica synchronization protocol. Dynamo employs

a gossip based distributed failure detection and membership protocol. Dynamo is a completely decentralized system with minimal need for manual administration. Storage nodes can be added and removed from Dynamo without requiring any manual partitioning or redistribution.

In the past year, Dynamo has been the underlying storage technology for a number of the core services in Amazon's e-commerce platform. It was able to scale to extreme peak loads efficiently without any downtime during the busy holiday shopping season. For example, the service that maintains shopping cart (Shopping Cart Service) served tens of millions requests that resulted in well over 3 million checkouts in a single day and the service that manages session state handled hundreds of thousands of concurrently active sessions.

The main contribution of this work for the research community is the evaluation of how different techniques can be combined to provide a single highly-available system. It demonstrates that an eventually-consistent storage system can be used in production with demanding applications. It also provides insight into the tuning of these techniques to meet the requirements of production systems with very strict performance demands.

The paper is structured as follows. Section 2 presents the background and Section 3 presents the related work. Section 4 presents the system design and Section 5 describes the implementation. Section 6 details the experiences and insights gained by running Dynamo in production and Section 7 concludes the paper. There are a number of places in this paper where additional information may have been appropriate but where protecting Amazon's business interests require us to reduce some level of detail. For this reason, the intra- and inter-datacenter latencies in section 6, the absolute request rates in section 6.2 and outage lengths and workloads in section 6.3 are provided through aggregate measures instead of absolute details.

2. BACKGROUND

Amazon's e-commerce platform is composed of hundreds of services that work in concert to deliver functionality ranging from recommendations to order fulfillment to fraud detection. Each service is exposed through a well defined interface and is accessible over the network. These services are hosted in an infrastructure that consists of tens of thousands of servers located across many data centers world-wide. Some of these services are stateless (i.e., services which aggregate responses from other services) and some are stateful (i.e., a service that generates its response by executing business logic on its state stored in persistent store).

Traditionally production systems store their state in relational databases. For many of the more common usage patterns of state persistence, however, a relational database is a solution that is far from ideal. Most of these services only store and retrieve data by primary key and do not require the complex querying and management functionality offered by an RDBMS. This excess functionality requires expensive hardware and highly skilled personnel for its operation, making it a very inefficient solution. In addition, the available replication technologies are limited and typically choose consistency over availability. Although many advances have been made in the recent years, it is still not easy to scale-out databases or use smart partitioning schemes for load balancing.

This paper describes Dynamo, a highly available data storage technology that addresses the needs of these important classes of services. Dynamo has a simple key/value interface, is highly available with a clearly defined consistency window, is efficient in its resource usage, and has a simple scale out scheme to address growth in data set size or request rates. Each service that uses Dynamo runs its own Dynamo instances.

2.1 System Assumptions and Requirements

The storage system for this class of services has the following requirements:

Query Model: simple read and write operations to a data item that is uniquely identified by a key. State is stored as binary objects (i.e., blobs) identified by unique keys. No operations span multiple data items and there is no need for relational schema. This requirement is based on the observation that a significant portion of Amazon's services can work with this simple query model and do not need any relational schema. Dynamo targets applications that need to store objects that are relatively small (usually less than 1 MB).

ACID Properties: ACID (*Atomicity, Consistency, Isolation, Durability*) is a set of properties that guarantee that database transactions are processed reliably. In the context of databases, a single logical operation on the data is called a transaction. Experience at Amazon has shown that data stores that provide ACID guarantees tend to have poor availability. This has been widely acknowledged by both the industry and academia [5]. Dynamo targets applications that operate with weaker consistency (the "C" in ACID) if this results in high availability. Dynamo does not provide any isolation guarantees and permits only single key updates.

Efficiency: The system needs to function on a commodity hardware infrastructure. In Amazon's platform, services have stringent latency requirements which are in general measured at the 99.9th percentile of the distribution. Given that state access plays a crucial role in service operation the storage system must be capable of meeting such stringent SLAs (see Section 2.2 below). Services must be able to configure Dynamo such that they consistently achieve their latency and throughput requirements. The tradeoffs are in performance, cost efficiency, availability, and durability guarantees.

Other Assumptions: Dynamo is used only by Amazon's internal services. Its operation environment is assumed to be non-hostile and there are no security related requirements such as authentication and authorization. Moreover, since each service uses its distinct instance of Dynamo, its initial design targets a scale of up to hundreds of storage hosts. We will discuss the scalability limitations of Dynamo and possible scalability related extensions in later sections.

2.2 Service Level Agreements (SLA)

To guarantee that the application can deliver its functionality in a bounded time, each and every dependency in the platform needs to deliver its functionality with even tighter bounds. Clients and services engage in a Service Level Agreement (SLA), a formally negotiated contract where a client and a service agree on several system-related characteristics, which most prominently include the client's expected request rate distribution for a particular API and the expected service latency under those conditions. An example of a simple SLA is a service guaranteeing that it will

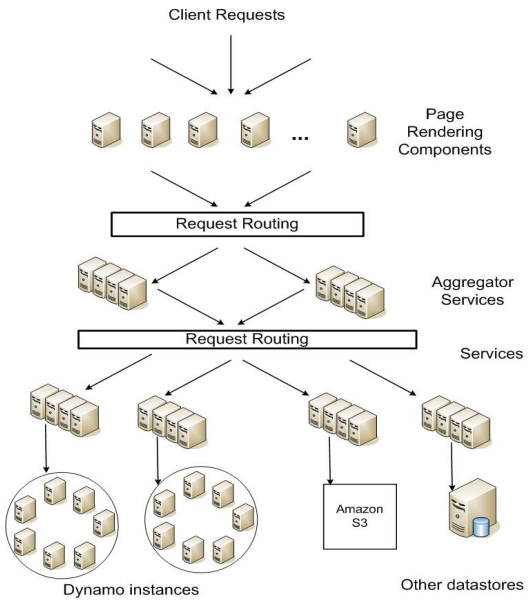


Figure 1: Service-oriented architecture of Amazon's platform

provide a response within 300ms for 99.9% of its requests for a peak client load of 500 requests per second.

In Amazon's decentralized service oriented infrastructure, SLAs play an important role. For example a page request to one of the e-commerce sites typically requires the rendering engine to construct its response by sending requests to over 150 services. These services often have multiple dependencies, which frequently are other services, and as such it is not uncommon for the call graph of an application to have more than one level. To ensure that the page rendering engine can maintain a clear bound on page delivery each service within the call chain must obey its performance contract.

Figure 1 shows an abstract view of the architecture of Amazon's platform, where dynamic web content is generated by page rendering components which in turn query many other services. A service can use different data stores to manage its state and these data stores are only accessible within its service boundaries. Some services act as aggregators by using several other services to produce a composite response. Typically, the aggregator services are stateless, although they use extensive caching.

A common approach in the industry for forming a performance oriented SLA is to describe it using average, median and expected variance. At Amazon we have found that these metrics are not good enough if the goal is to build a system where **all** customers have a good experience, rather than just the majority. For example if extensive personalization techniques are used then customers with longer histories require more processing which impacts performance at the high-end of the distribution. An SLA stated in terms of mean or median response times will not address the performance of this important customer segment. To address this issue, at Amazon, SLAs are expressed and measured at the 99.9th percentile of the distribution. The choice for 99.9% over an even higher percentile has been made based on a cost-benefit analysis which demonstrated a significant increase in cost to improve performance that much. Experiences with Amazon's production systems have shown that this approach provides a better overall experience compared to those systems that meet SLAs defined based on the mean or median.

In this paper there are many references to this 99.9th percentile of distributions, which reflects Amazon engineers' relentless focus on performance from the perspective of the customers' experience. Many papers report on averages, so these are included where it makes sense for comparison purposes. Nevertheless, Amazon's engineering and optimization efforts are not focused on averages. Several techniques, such as the load balanced selection of write coordinators, are purely targeted at controlling performance at the 99.9th percentile.

Storage systems often play an important role in establishing a service's SLA, especially if the business logic is relatively lightweight, as is the case for many Amazon services. State management then becomes the main component of a service's SLA. One of the main design considerations for Dynamo is to give services control over their system properties, such as durability and consistency, and to let services make their own tradeoffs between functionality, performance and cost-effectiveness.

2.3 Design Considerations

Data replication algorithms used in commercial systems traditionally perform synchronous replica coordination in order to provide a strongly consistent data access interface. To achieve this level of consistency, these algorithms are forced to tradeoff the availability of the data under certain failure scenarios. For instance, rather than dealing with the uncertainty of the correctness of an answer, the data is made unavailable until it is absolutely certain that it is correct. From the very early replicated database works, it is well known that when dealing with the possibility of network failures, strong consistency and high data availability cannot be achieved simultaneously [2, 11]. As such systems and applications need to be aware which properties can be achieved under which conditions.

For systems prone to server and network failures, availability can be increased by using optimistic replication techniques, where changes are allowed to propagate to replicas in the background, and concurrent, disconnected work is tolerated. The challenge with this approach is that it can lead to conflicting changes which must be detected and resolved. This process of conflict resolution introduces two problems: when to resolve them and who resolves them. Dynamo is designed to be an eventually consistent data store; that is all updates reach all replicas eventually.

An important design consideration is to decide *when* to perform the process of resolving update conflicts, i.e., whether conflicts should be resolved during reads or writes. Many traditional data stores execute conflict resolution during writes and keep the read complexity simple [7]. In such systems, writes may be rejected if the data store cannot reach all (or a majority of) the replicas at a given time. On the other hand, Dynamo targets the design space of an "always writeable" data store (i.e., a data store that is highly available for writes). For a number of Amazon services, rejecting customer updates could result in a poor customer experience. For instance, the shopping cart service must allow customers to add and remove items from their shopping cart even amidst network and server failures. This requirement forces us to push the complexity of conflict resolution to the reads in order to ensure that writes are never rejected.

The next design choice is *who* performs the process of conflict resolution. This can be done by the data store or the application. If conflict resolution is done by the data store, its choices are rather limited. In such cases, the data store can only use simple policies, such as "last write wins" [22], to resolve conflicting updates. On the other hand, since the application is aware of the data schema it can decide on the conflict resolution method that is best suited for its client's experience. For instance, the application that maintains customer shopping carts can choose to "merge" the conflicting versions and return a single unified shopping cart. Despite this flexibility, some application developers may not want to write their own conflict resolution mechanisms and choose to push it down to the data store, which in turn chooses a simple policy such as "last write wins".

Other key principles embraced in the design are:

Incremental scalability: Dynamo should be able to scale out one storage host (henceforth, referred to as "*node"*) at a time, with minimal impact on both operators of the system and the system itself.

Symmetry: Every node in Dynamo should have the same set of responsibilities as its peers; there should be no distinguished node or nodes that take special roles or extra set of responsibilities. In our experience, symmetry simplifies the process of system provisioning and maintenance.

Decentralization: An extension of symmetry, the design should favor decentralized peer-to-peer techniques over centralized control. In the past, centralized control has resulted in outages and the goal is to avoid it as much as possible. This leads to a simpler, more scalable, and more available system.

Heterogeneity: The system needs to be able to exploit heterogeneity in the infrastructure it runs on. e.g. the work distribution must be proportional to the capabilities of the individual servers. This is essential in adding new nodes with higher capacity without having to upgrade all hosts at once.

3. RELATED WORK

3.1 Peer to Peer Systems

There are several peer-to-peer (P2P) systems that have looked at the problem of data storage and distribution. The first generation of P2P systems, such as Freenet and Gnutella[1], were predominantly used as file sharing systems. These were examples of unstructured P2P networks where the overlay links between peers were established arbitrarily. In these networks, a search query is usually flooded through the network to find as many peers as possible that share the data. P2P systems evolved to the next generation into what is widely known as structured P2P networks. These networks employ a globally consistent protocol to ensure that any node can efficiently route a search query to some peer that has the desired data. Systems like Pastry [16] and Chord [20] use routing mechanisms to ensure that queries can be answered within a bounded number of hops. To reduce the additional latency introduced by multi-hop routing, some P2P systems (e.g., [14]) employ O(1) routing where each peer maintains enough routing information locally so that it can route requests (to access a data item) to the appropriate peer within a constant number of hops.

[1] http://freenetproject.org/, http://www.gnutella.org

Various storage systems, such as Oceanstore [9] and PAST [17] were built on top of these routing overlays. Oceanstore provides a global, transactional, persistent storage service that supports serialized updates on widely replicated data. To allow for concurrent updates while avoiding many of the problems inherent with wide-area locking, it uses an update model based on conflict resolution. Conflict resolution was introduced in [21] to reduce the number of transaction aborts. Oceanstore resolves conflicts by processing a series of updates, choosing a total order among them, and then applying them atomically in that order. It is built for an environment where the data is replicated on an untrusted infrastructure. By comparison, PAST provides a simple abstraction layer on top of Pastry for persistent and immutable objects. It assumes that the application can build the necessary storage semantics (such as mutable files) on top of it.

3.2 Distributed File Systems and Databases

Distributing data for performance, availability and durability has been widely studied in the file system and database systems community. Compared to P2P storage systems that only support flat namespaces, distributed file systems typically support hierarchical namespaces. Systems like Ficus [15] and Coda [19] replicate files for high availability at the expense of consistency. Update conflicts are typically managed using specialized conflict resolution procedures. The Farsite system [1] is a distributed file system that does not use any centralized server like NFS. Farsite achieves high availability and scalability using replication. The Google File System [6] is another distributed file system built for hosting the state of Google's internal applications. GFS uses a simple design with a single master server for hosting the entire metadata and where the data is split into chunks and stored in chunkservers. Bayou is a distributed relational database system that allows disconnected operations and provides eventual data consistency [21].

Among these systems, Bayou, Coda and Ficus allow disconnected operations and are resilient to issues such as network partitions and outages. These systems differ on their conflict resolution procedures. For instance, Coda and Ficus perform system level conflict resolution and Bayou allows application level resolution. All of them, however, guarantee eventual consistency. Similar to these systems, Dynamo allows read and write operations to continue even during network partitions and resolves updated conflicts using different conflict resolution mechanisms. Distributed block storage systems like FAB [18] split large size objects into smaller blocks and stores each block in a highly available manner. In comparison to these systems, a key-value store is more suitable in this case because: (a) it is intended to store relatively small objects (size < 1M) and (b) key-value stores are easier to configure on a per-application basis. Antiquity is a wide-area distributed storage system designed to handle multiple server failures [23]. It uses a secure log to preserve data integrity, replicates each log on multiple servers for durability, and uses Byzantine fault tolerance protocols to ensure data consistency. In contrast to Antiquity, Dynamo does not focus on the problem of data integrity and security and is built for a trusted environment. Bigtable is a distributed storage system for managing structured data. It maintains a sparse, multi-dimensional sorted map and allows applications to access their data using multiple attributes [2]. Compared to Bigtable, Dynamo targets applications that require only key/value access with primary focus on high availability where updates are not rejected even in the wake of network partitions or server failures.

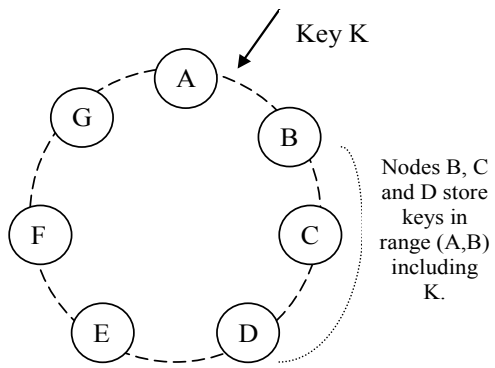


Figure 2: Partitioning and replication of keys in Dynamo ring.

Traditional replicated relational database systems focus on the problem of guaranteeing strong consistency to replicated data. Although strong consistency provides the application writer a convenient programming model, these systems are limited in scalability and availability [7]. These systems are not capable of handling network partitions because they typically provide strong consistency guarantees.

3.3 Discussion

Dynamo differs from the aforementioned decentralized storage systems in terms of its target requirements. First, Dynamo is targeted mainly at applications that need an "always writeable" data store where no updates are rejected due to failures or concurrent writes. This is a crucial requirement for many Amazon applications. Second, as noted earlier, Dynamo is built for an infrastructure within a single administrative domain where all nodes are assumed to be trusted. Third, applications that use Dynamo do not require support for hierarchical namespaces (a norm in many file systems) or complex relational schema (supported by traditional databases). Fourth, Dynamo is built for latency sensitive applications that require at least 99.9% of read and write operations to be performed within a few hundred milliseconds. To meet these stringent latency requirements, it was imperative for us to avoid routing requests through multiple nodes (which is the typical design adopted by several distributed hash table systems such as Chord and Pastry). This is because multi-hop routing increases variability in response times, thereby increasing the latency at higher percentiles. Dynamo can be characterized as a zero-hop DHT, where each node maintains enough routing information locally to route a request to the appropriate node directly.

4. SYSTEM ARCHITECTURE

The architecture of a storage system that needs to operate in a production setting is complex. In addition to the actual data persistence component, the system needs to have scalable and robust solutions for load balancing, membership and failure detection, failure recovery, replica synchronization, overload handling, state transfer, concurrency and job scheduling, request marshalling, request routing, system monitoring and alarming, and configuration management. Describing the details of each of the solutions is not possible, so this paper focuses on the core distributed systems techniques used in Dynamo: partitioning, replication, versioning, membership, failure handling and scaling.

Table 1: Summary of techniques used in *Dynamo* and their advantages.

Problem	Technique	Advantage
Partitioning	Consistent Hashing	Incremental Scalability
High Availability for writes	Vector clocks with reconciliation during reads	Version size is decoupled from update rates.
Handling temporary failures	Sloppy Quorum and hinted handoff	Provides high availability and durability guarantee when some of the replicas are not available.
Recovering from permanent failures	Anti-entropy using Merkle trees	Synchronizes divergent replicas in the background.
Membership and failure detection	Gossip-based membership protocol and failure detection.	Preserves symmetry and avoids having a centralized registry for storing membership and node liveness information.

Table 1 presents a summary of the list of techniques Dynamo uses and their respective advantages.

4.1 System Interface

Dynamo stores objects associated with a key through a simple interface; it exposes two operations: get() and put(). The get(*key*) operation locates the object replicas associated with the *key* in the storage system and returns a single object or a list of objects with conflicting versions along with a *context*. The put(*key, context, object*) operation determines where the replicas of the *object* should be placed based on the associated *key*, and writes the replicas to disk. The *context* encodes system metadata about the object that is opaque to the caller and includes information such as the version of the object. The context information is stored along with the object so that the system can verify the validity of the context object supplied in the put request.

Dynamo treats both the key and the object supplied by the caller as an opaque array of bytes. It applies a MD5 hash on the key to generate a 128-bit identifier, which is used to determine the storage nodes that are responsible for serving the key.

4.2 Partitioning Algorithm

One of the key design requirements for Dynamo is that it must scale incrementally. This requires a mechanism to dynamically partition the data over the set of nodes (i.e., storage hosts) in the system. Dynamo's partitioning scheme relies on consistent hashing to distribute the load across multiple storage hosts. In consistent hashing [10], the output range of a hash function is treated as a fixed circular space or "ring" (i.e. the largest hash value wraps around to the smallest hash value). Each node in the system is assigned a random value within this space which represents its "position" on the ring. Each data item identified by a key is assigned to a node by hashing the data item's key to yield its position on the ring, and then walking the ring clockwise to find the first node with a position larger than the item's position.

Thus, each node becomes responsible for the region in the ring between it and its predecessor node on the ring. The principle advantage of consistent hashing is that departure or arrival of a node only affects its immediate neighbors and other nodes remain unaffected.

The basic consistent hashing algorithm presents some challenges. First, the random position assignment of each node on the ring leads to non-uniform data and load distribution. Second, the basic algorithm is oblivious to the heterogeneity in the performance of nodes. To address these issues, Dynamo uses a variant of consistent hashing (similar to the one used in [10, 20]): instead of mapping a node to a single point in the circle, each node gets assigned to multiple points in the ring. To this end, Dynamo uses the concept of "virtual nodes". A virtual node looks like a single node in the system, but each node can be responsible for more than one virtual node. Effectively, when a new node is added to the system, it is assigned multiple positions (henceforth, "tokens") in the ring. The process of fine-tuning Dynamo's partitioning scheme is discussed in Section 6.

Using virtual nodes has the following advantages:

- If a node becomes unavailable (due to failures or routine maintenance), the load handled by this node is evenly dispersed across the remaining available nodes.
- When a node becomes available again, or a new node is added to the system, the newly available node accepts a roughly equivalent amount of load from each of the other available nodes.
- The number of virtual nodes that a node is responsible can decided based on its capacity, accounting for heterogeneity in the physical infrastructure.

4.3 Replication

To achieve high availability and durability, Dynamo replicates its data on multiple hosts. Each data item is replicated at N hosts, where N is a parameter configured *"per-instance"*. Each key, *k*, is assigned to a coordinator node (described in the previous section). The coordinator is in charge of the replication of the data items that fall within its range. In addition to locally storing each key within its range, the coordinator replicates these keys at the N-1 clockwise successor nodes in the ring. This results in a system where each node is responsible for the region of the ring between it and its Nth predecessor. In Figure 2, node B replicates the key *k* at nodes C and D in addition to storing it locally. Node D will store the keys that fall in the ranges (A, B], (B, C], and (C, D].

The list of nodes that is responsible for storing a particular key is called the *preference list*. The system is designed, as will be explained in Section 4.8, so that every node in the system can determine which nodes should be in this list for any particular key. To account for node failures, preference list contains more than N nodes. Note that with the use of virtual nodes, it is possible that the first N successor positions for a particular key may be owned by less than N distinct physical nodes (i.e. a node may hold more than one of the first N positions). To address this, the preference list for a key is constructed by skipping positions in the ring to ensure that the list contains only distinct physical nodes.

4.4 Data Versioning

Dynamo provides eventual consistency, which allows for updates to be propagated to all replicas asynchronously. A put() call may return to its caller before the update has been applied at all the replicas, which can result in scenarios where a subsequent get() operation may return an object that does not have the latest updates.. If there are no failures then there is a bound on the update propagation times. However, under certain failure scenarios (e.g., server outages or network partitions), updates may not arrive at all replicas for an extended period of time.

There is a category of applications in Amazon's platform that can tolerate such inconsistencies and can be constructed to operate under these conditions. For example, the shopping cart application requires that an "*Add to Cart"* operation can never be forgotten or rejected. If the most recent state of the cart is unavailable, and a user makes changes to an older version of the cart, that change is still meaningful and should be preserved. But at the same time it shouldn't supersede the currently unavailable state of the cart, which itself may contain changes that should be preserved. Note that both "*add to cart*" and "*delete item from cart*" operations are translated into put requests to Dynamo. When a customer wants to add an item to (or remove from) a shopping cart and the latest version is not available, the item is added to (or removed from) the older version and the divergent versions are reconciled later.

In order to provide this kind of guarantee, Dynamo treats the result of each modification as a new and immutable version of the data. It allows for multiple versions of an object to be present in the system at the same time. Most of the time, new versions subsume the previous version(s), and the system itself can determine the authoritative version (syntactic reconciliation). However, version branching may happen, in the presence of failures combined with concurrent updates, resulting in conflicting versions of an object. In these cases, the system cannot reconcile the multiple versions of the same object and the client must perform the reconciliation in order to *collapse* multiple branches of data evolution back into one (semantic reconciliation). A typical example of a collapse operation is "merging" different versions of a customer's shopping cart. Using this reconciliation mechanism, an "add to cart" operation is never lost. However, deleted items can resurface.

It is important to understand that certain failure modes can potentially result in the system having not just two but several versions of the same data. Updates in the presence of network partitions and node failures can potentially result in an object having distinct version sub-histories, which the system will need to reconcile in the future. This requires us to design applications that explicitly acknowledge the possibility of multiple versions of the same data (in order to never lose any updates).

Dynamo uses vector clocks [12] in order to capture causality between different versions of the same object. A vector clock is effectively a list of (node, counter) pairs. One vector clock is associated with every version of every object. One can determine whether two versions of an object are on parallel branches or have a causal ordering, by examine their vector clocks. If the counters on the first object's clock are less-than-or-equal to all of the nodes in the second clock, then the first is an ancestor of the second and can be forgotten. Otherwise, the two changes are considered to be in conflict and require reconciliation.

In Dynamo, when a client wishes to update an object, it must specify which version it is updating. This is done by passing the context it obtained from an earlier read operation, which contains the vector clock information. Upon processing a read request, if

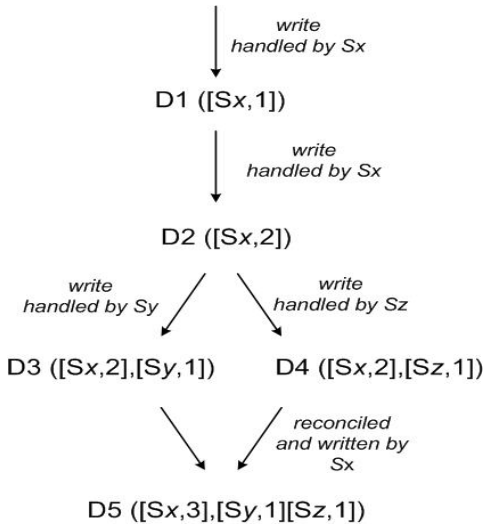


Figure 3: Version evolution of an object over time.

Dynamo has access to multiple branches that cannot be syntactically reconciled, it will return all the objects at the leaves, with the corresponding version information in the context. An update using this context is considered to have reconciled the divergent versions and the branches are collapsed into a single new version.

To illustrate the use of vector clocks, let us consider the example shown in Figure 3. A client writes a new object. The node (say Sx) that handles the write for this key increases its sequence number and uses it to create the data's vector clock. The system now has the object D1 and its associated clock [(Sx, 1)]. The client updates the object. Assume the same node handles this request as well. The system now also has object D2 and its associated clock [(Sx, 2)]. D2 *descends* from D1 and therefore over-writes D1, however there may be replicas of D1 lingering at nodes that have not yet seen D2. Let us assume that the same client updates the object again and a different server (say Sy) handles the request. The system now has data D3 and its associated clock [(Sx, 2), (Sy, 1)].

Next assume a different client reads D2 and then tries to update it, and another node (say Sz) does the write. The system now has D4 (descendant of D2) whose version clock is [(Sx, 2), (Sz, 1)]. A node that is aware of D1 or D2 could determine, upon receiving D4 and its clock, that D1 and D2 are overwritten by the new data and can be garbage collected. A node that is aware of D3 and receives D4 will find that there is no causal relation between them. In other words, there are changes in D3 and D4 that are not reflected in each other. Both versions of the data must be kept and presented to a client (upon a read) for semantic reconciliation.

Now assume some client reads both D3 and D4 (the context will reflect that both values were found by the read). The read's context is a summary of the clocks of D3 and D4, namely [(Sx, 2), (Sy, 1), (Sz, 1)]. If the client performs the reconciliation and node Sx coordinates the write, Sx will update its sequence number in the clock. The new data D5 will have the following clock: [(Sx, 3), (Sy, 1), (Sz, 1)].

A possible issue with vector clocks is that the size of vector clocks may grow if many servers coordinate the writes to an object. In practice, this is not likely because the writes are usually handled by one of the top N nodes in the preference list. In case of network partitions or multiple server failures, write requests may be handled by nodes that are not in the top N nodes in the preference list causing the size of vector clock to grow. In these scenarios, it is desirable to limit the size of vector clock. To this end, Dynamo employs the following clock truncation scheme: Along with each (node, counter) pair, Dynamo stores a timestamp that indicates the last time the node updated the data item. When the number of (node, counter) pairs in the vector clock reaches a threshold (say 10), the oldest pair is removed from the clock. Clearly, this truncation scheme can lead to inefficiencies in reconciliation as the descendant relationships cannot be derived accurately. However, this problem has not surfaced in production and therefore this issue has not been thoroughly investigated.

4.5 Execution of get () and put () operations

Any storage node in Dynamo is eligible to receive client get and put operations for any key. In this section, for sake of simplicity, we describe how these operations are performed in a failure-free environment and in the subsequent section we describe how read and write operations are executed during failures.

Both get and put operations are invoked using Amazon's infrastructure-specific request processing framework over HTTP. There are two strategies that a client can use to select a node: (1) route its request through a generic load balancer that will select a node based on load information, or (2) use a partition-aware client library that routes requests directly to the appropriate coordinator nodes. The advantage of the first approach is that the client does not have to link any code specific to Dynamo in its application, whereas the second strategy can achieve lower latency because it skips a potential forwarding step.

A node handling a read or write operation is known as the *coordinator*. Typically, this is the first among the top N nodes in the preference list. If the requests are received through a load balancer, requests to access a key may be routed to any random node in the ring. In this scenario, the node that receives the request will not coordinate it if the node is not in the top N of the requested key's preference list. Instead, that node will forward the request to the first among the top N nodes in the preference list.

Read and write operations involve the first N healthy nodes in the preference list, skipping over those that are down or inaccessible. When all nodes are healthy, the top N nodes in a key's preference list are accessed. When there are node failures or network partitions, nodes that are lower ranked in the preference list are accessed.

To maintain consistency among its replicas, Dynamo uses a consistency protocol similar to those used in quorum systems. This protocol has two key configurable values: R and W. R is the minimum number of nodes that must participate in a successful read operation. W is the minimum number of nodes that must participate in a successful write operation. Setting R and W such that R + W > N yields a quorum-like system. In this model, the latency of a get (or put) operation is dictated by the slowest of the R (or W) replicas. For this reason, R and W are usually configured to be less than N, to provide better latency.

Upon receiving a put() request for a key, the coordinator generates the vector clock for the new version and writes the new version locally. The coordinator then sends the new version (along with

the new vector clock) to the N highest-ranked reachable nodes. If at least W-1 nodes respond then the write is considered successful.

Similarly, for a get() request, the coordinator requests all existing versions of data for that key from the N highest-ranked reachable nodes in the preference list for that key, and then waits for R responses before returning the result to the client. If the coordinator ends up gathering multiple versions of the data, it returns all the versions it deems to be causally unrelated. The divergent versions are then reconciled and the reconciled version superseding the current versions is written back.

4.6 Handling Failures: Hinted Handoff

If Dynamo used a traditional quorum approach it would be unavailable during server failures and network partitions, and would have reduced durability even under the simplest of failure conditions. To remedy this it does not enforce strict quorum membership and instead it uses a "sloppy quorum"; all read and write operations are performed on the first N *healthy* nodes from the preference list, which may not always be the first N nodes encountered while walking the consistent hashing ring.

Consider the example of Dynamo configuration given in Figure 2 with N=3. In this example, if node A is temporarily down or unreachable during a write operation then a replica that would normally have lived on A will now be sent to node D. This is done to maintain the desired availability and durability guarantees. The replica sent to D will have a hint in its metadata that suggests which node was the intended recipient of the replica (in this case A). Nodes that receive hinted replicas will keep them in a separate local database that is scanned periodically. Upon detecting that A has recovered, D will attempt to deliver the replica to A. Once the transfer succeeds, D may delete the object from its local store without decreasing the total number of replicas in the system.

Using hinted handoff, Dynamo ensures that the read and write operations are not failed due to temporary node or network failures. Applications that need the highest level of availability can set W to 1, which ensures that a write is accepted as long as a single node in the system has durably written the key it to its local store. Thus, the write request is only rejected if all nodes in the system are unavailable. However, in practice, most Amazon services in production set a higher W to meet the desired level of durability. A more detailed discussion of configuring N, R and W follows in section 6.

It is imperative that a highly available storage system be capable of handling the failure of an entire data center(s). Data center failures happen due to power outages, cooling failures, network failures, and natural disasters. Dynamo is configured such that each object is replicated across multiple data centers. In essence, the preference list of a key is constructed such that the storage nodes are spread across multiple data centers. These datacenters are connected through high speed network links. This scheme of replicating across multiple datacenters allows us to handle entire data center failures without a data outage.

4.7 Handling permanent failures: Replica synchronization

Hinted handoff works best if the system membership churn is low and node failures are transient. There are scenarios under which hinted replicas become unavailable before they can be returned to the original replica node. To handle this and other threats to durability, Dynamo implements an anti-entropy (replica synchronization) protocol to keep the replicas synchronized.

To detect the inconsistencies between replicas faster and to minimize the amount of transferred data, Dynamo uses Merkle trees [13]. A Merkle tree is a hash tree where leaves are hashes of the values of individual keys. Parent nodes higher in the tree are hashes of their respective children. The principal advantage of Merkle tree is that each branch of the tree can be checked independently without requiring nodes to download the entire tree or the entire data set. Moreover, Merkle trees help in reducing the amount of data that needs to be transferred while checking for inconsistencies among replicas. For instance, if the hash values of the root of two trees are equal, then the values of the leaf nodes in the tree are equal and the nodes require no synchronization. If not, it implies that the values of some replicas are different. In such cases, the nodes may exchange the hash values of children and the process continues until it reaches the leaves of the trees, at which point the hosts can identify the keys that are "out of sync". Merkle trees minimize the amount of data that needs to be transferred for synchronization and reduce the number of disk reads performed during the anti-entropy process.

Dynamo uses Merkle trees for anti-entropy as follows: Each node maintains a separate Merkle tree for each key range (the set of keys covered by a virtual node) it hosts. This allows nodes to compare whether the keys within a key range are up-to-date. In this scheme, two nodes exchange the root of the Merkle tree corresponding to the key ranges that they host in common. Subsequently, using the tree traversal scheme described above the nodes determine if they have any differences and perform the appropriate synchronization action. The disadvantage with this scheme is that many key ranges change when a node joins or leaves the system thereby requiring the tree(s) to be recalculated. This issue is addressed, however, by the refined partitioning scheme described in Section 6.2.

4.8 Membership and Failure Detection

4.8.1 Ring Membership

In Amazon's environment node outages (due to failures and maintenance tasks) are often transient but may last for extended intervals. A node outage rarely signifies a permanent departure and therefore should not result in rebalancing of the partition assignment or repair of the unreachable replicas. Similarly, manual error could result in the unintentional startup of new Dynamo nodes. For these reasons, it was deemed appropriate to use an explicit mechanism to initiate the addition and removal of nodes from a Dynamo ring. An administrator uses a command line tool or a browser to connect to a Dynamo node and issue a membership change to join a node to a ring or remove a node from a ring. The node that serves the request writes the membership change and its time of issue to persistent store. The membership changes form a history because nodes can be removed and added back multiple times. A gossip-based protocol propagates membership changes and maintains an eventually consistent view of membership. Each node contacts a peer chosen at random every second and the two nodes efficiently reconcile their persisted membership change histories.

When a node starts for the first time, it chooses its set of tokens (virtual nodes in the consistent hash space) and maps nodes to their respective token sets. The mapping is persisted on disk and

initially contains only the local node and token set. The mappings stored at different Dynamo nodes are reconciled during the same communication exchange that reconciles the membership change histories. Therefore, partitioning and placement information also propagates via the gossip-based protocol and each storage node is aware of the token ranges handled by its peers. This allows each node to forward a key's read/write operations to the right set of nodes directly.

4.8.2 External Discovery

The mechanism described above could temporarily result in a logically partitioned Dynamo ring. For example, the administrator could contact node A to join A to the ring, then contact node B to join B to the ring. In this scenario, nodes A and B would each consider itself a member of the ring, yet neither would be immediately aware of the other. To prevent logical partitions, some Dynamo nodes play the role of seeds. Seeds are nodes that are discovered via an external mechanism and are known to all nodes. Because all nodes eventually reconcile their membership with a seed, logical partitions are highly unlikely. Seeds can be obtained either from static configuration or from a configuration service. Typically seeds are fully functional nodes in the Dynamo ring.

4.8.3 Failure Detection

Failure detection in Dynamo is used to avoid attempts to communicate with unreachable peers during get() and put() operations and when transferring partitions and hinted replicas. For the purpose of avoiding failed attempts at communication, a purely local notion of failure detection is entirely sufficient: node A may consider node B failed if node B does not respond to node A's messages (even if B is responsive to node C's messages). In the presence of a steady rate of client requests generating inter-node communication in the Dynamo ring, a node A quickly discovers that a node B is unresponsive when B fails to respond to a message; Node A then uses alternate nodes to service requests that map to B's partitions; A periodically retries B to check for the latter's recovery. In the absence of client requests to drive traffic between two nodes, neither node really needs to know whether the other is reachable and responsive.

Decentralized failure detection protocols use a simple gossip-style protocol that enable each node in the system to learn about the arrival (or departure) of other nodes. For detailed information on decentralized failure detectors and the parameters affecting their accuracy, the interested reader is referred to [8]. Early designs of Dynamo used a decentralized failure detector to maintain a globally consistent view of failure state. Later it was determined that the explicit node join and leave methods obviates the need for a global view of failure state. This is because nodes are notified of permanent node additions and removals by the explicit node join and leave methods and temporary node failures are detected by the individual nodes when they fail to communicate with others (while forwarding requests).

4.9 Adding/Removing Storage Nodes

When a new node (say X) is added into the system, it gets assigned a number of tokens that are randomly scattered on the ring. For every key range that is assigned to node X, there may be a number of nodes (less than or equal to N) that are currently in charge of handling keys that fall within its token range. Due to the allocation of key ranges to X, some existing nodes no longer have to some of their keys and these nodes transfer those keys to X. Let us consider a simple bootstrapping scenario where node X is added to the ring shown in Figure 2 between A and B. When X is added to the system, it is in charge of storing keys in the ranges (F, G], (G, A] and (A, X]. As a consequence, nodes B, C and D no longer have to store the keys in these respective ranges. Therefore, nodes B, C, and D will offer to and upon confirmation from X transfer the appropriate set of keys. When a node is removed from the system, the reallocation of keys happens in a reverse process.

Operational experience has shown that this approach distributes the load of key distribution uniformly across the storage nodes, which is important to meet the latency requirements and to ensure fast bootstrapping. Finally, by adding a confirmation round between the source and the destination, it is made sure that the destination node does not receive any duplicate transfers for a given key range.

5. IMPLEMENTATION

In Dynamo, each storage node has three main software components: request coordination, membership and failure detection, and a local persistence engine. All these components are implemented in Java.

Dynamo's local persistence component allows for different storage engines to be plugged in. Engines that are in use are Berkeley Database (BDB) Transactional Data Store[2], BDB Java Edition, MySQL, and an in-memory buffer with persistent backing store. The main reason for designing a pluggable persistence component is to choose the storage engine best suited for an application's access patterns. For instance, BDB can handle objects typically in the order of tens of kilobytes whereas MySQL can handle objects of larger sizes. Applications choose Dynamo's local persistence engine based on their object size distribution. The majority of Dynamo's production instances use BDB Transactional Data Store.

The request coordination component is built on top of an event-driven messaging substrate where the message processing pipeline is split into multiple stages similar to the SEDA architecture [24]. All communications are implemented using Java NIO channels. The coordinator executes the read and write requests on behalf of clients by collecting data from one or more nodes (in the case of reads) or storing data at one or more nodes (for writes). Each client request results in the creation of a state machine on the node that received the client request. The state machine contains all the logic for identifying the nodes responsible for a key, sending the requests, waiting for responses, potentially doing retries, processing the replies and packaging the response to the client. Each state machine instance handles exactly one client request. For instance, a read operation implements the following state machine: (i) send read requests to the nodes, (ii) wait for minimum number of required responses, (iii) if too few replies were received within a given time bound, fail the request, (iv) otherwise gather all the data versions and determine the ones to be returned and (v) if versioning is enabled, perform syntactic reconciliation and generate an opaque write context that contains the vector clock that subsumes all the remaining versions. For the sake of brevity the failure handling and retry states are left out.

After the read response has been returned to the caller the state

[2] http://www.oracle.com/database/berkeley-db.html

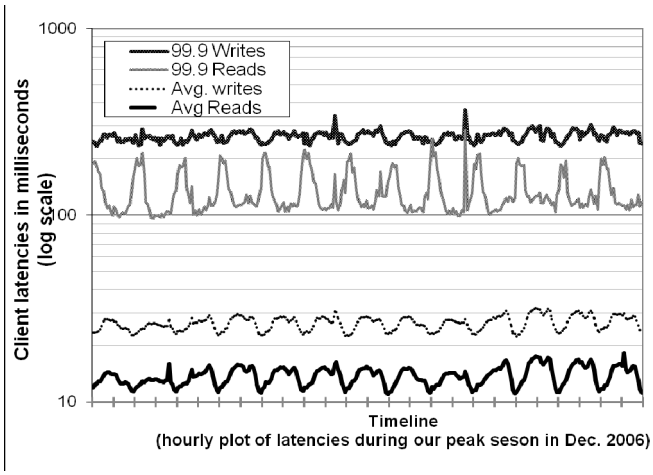


Figure 4: Average and 99.9 percentiles of latencies for read and write requests during our peak request season of December 2006. The intervals between consecutive ticks in the x-axis correspond to 12 hours. Latencies follow a diurnal pattern similar to the request rate and 99.9 percentile latencies are an order of magnitude higher than averages

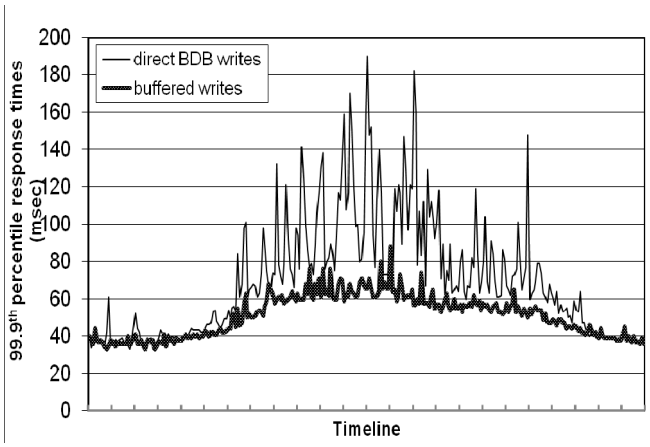


Figure 5: Comparison of performance of 99.9th percentile latencies for buffered vs. non-buffered writes over a period of 24 hours. The intervals between consecutive ticks in the x-axis correspond to one hour.

machine waits for a small period of time to receive any outstanding responses. If stale versions were returned in any of the responses, the coordinator updates those nodes with the latest version. This process is called *read repair* because it repairs replicas that have missed a recent update at an opportunistic time and relieves the anti-entropy protocol from having to do it.

As noted earlier, write requests are coordinated by one of the top N nodes in the preference list. Although it is desirable always to have the first node among the top N to coordinate the writes thereby serializing all writes at a single location, this approach has led to uneven load distribution resulting in SLA violations. This is because the request load is not uniformly distributed across objects. To counter this, any of the top N nodes in the preference list is allowed to coordinate the writes. In particular, since each write usually follows a read operation, the coordinator for a write is chosen to be the node that replied fastest to the previous read operation which is stored in the context information of the request. This optimization enables us to pick the node that has the data that was read by the preceding read operation thereby increasing the chances of getting "read-your-writes" consistency. It also reduces variability in the performance of the request handling which improves the performance at the 99.9 percentile.

6. EXPERIENCES & LESSONS LEARNED

Dynamo is used by several services with different configurations. These instances differ by their version reconciliation logic, and read/write quorum characteristics. The following are the main patterns in which Dynamo is used:

- *Business logic specific reconciliation:* This is a popular use case for Dynamo. Each data object is replicated across multiple nodes. In case of divergent versions, the client application performs its own reconciliation logic. The shopping cart service discussed earlier is a prime example of this category. Its business logic reconciles objects by merging different versions of a customer's shopping cart.
- *Timestamp based reconciliation:* This case differs from the previous one only in the reconciliation mechanism. In case of divergent versions, Dynamo performs simple timestamp based reconciliation logic of "last write wins"; i.e., the object with the largest physical timestamp value is chosen as the correct version. The service that maintains customer's session information is a good example of a service that uses this mode.
- *High performance read engine:* While Dynamo is built to be an "always writeable" data store, a few services are tuning its quorum characteristics and using it as a high performance read engine. Typically, these services have a high read request rate and only a small number of updates. In this configuration, typically R is set to be 1 and W to be N. For these services, Dynamo provides the ability to partition and replicate their data across multiple nodes thereby offering incremental scalability. Some of these instances function as the authoritative persistence cache for data stored in more heavy weight backing stores. Services that maintain product catalog and promotional items fit in this category.

The main advantage of Dynamo is that its client applications can tune the values of N, R and W to achieve their desired levels of performance, availability and durability. For instance, the value of N determines the durability of each object. A typical value of N used by Dynamo's users is 3.

The values of W and R impact object availability, durability and consistency. For instance, if W is set to 1, then the system will never reject a write request as long as there is at least one node in the system that can successfully process a write request. However, low values of W and R can increase the risk of inconsistency as write requests are deemed successful and returned to the clients even if they are not processed by a majority of the replicas. This also introduces a vulnerability window for durability when a write request is successfully returned to the client even though it has been persisted at only a small number of nodes.

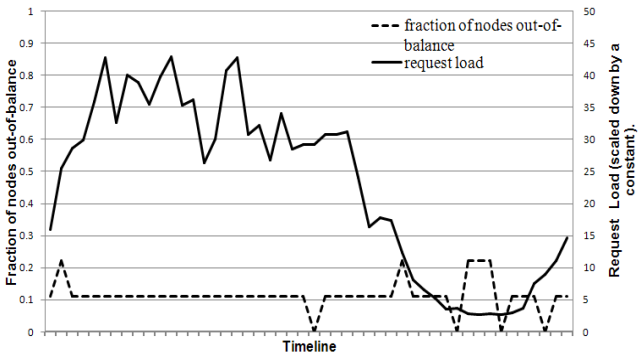


Figure 6: Fraction of nodes that are out-of-balance (i.e., nodes whose request load is above a certain threshold from the average system load) and their corresponding request load. The interval between ticks in x-axis corresponds to a time period of 30 minutes.

Traditional wisdom holds that durability and availability go hand-in-hand. However, this is not necessarily true here. For instance, the vulnerability window for durability can be decreased by increasing W. This may increase the probability of rejecting requests (thereby decreasing availability) because more storage hosts need to be alive to process a write request.

The common (N,R,W) configuration used by several instances of Dynamo is (3,2,2). These values are chosen to meet the necessary levels of performance, durability, consistency, and availability SLAs.

All the measurements presented in this section were taken on a live system operating with a configuration of (3,2,2) and running a couple hundred nodes with homogenous hardware configurations. As mentioned earlier, each instance of Dynamo contains nodes that are located in multiple datacenters. These datacenters are typically connected through high speed network links. Recall that to generate a successful get (or put) response R (or W) nodes need to respond to the coordinator. Clearly, the network latencies between datacenters affect the response time and the nodes (and their datacenter locations) are chosen such that the applications target SLAs are met.

6.1 Balancing Performance and Durability

While Dynamo's principle design goal is to build a highly available data store, performance is an equally important criterion in Amazon's platform. As noted earlier, to provide a consistent customer experience, Amazon's services set their performance targets at higher percentiles (such as the 99.9th or 99.99th percentiles). A typical SLA required of services that use Dynamo is that 99.9% of the read and write requests execute within 300ms.

Since Dynamo is run on standard commodity hardware components that have far less I/O throughput than high-end enterprise servers, providing consistently high performance for read and write operations is a non-trivial task. The involvement of multiple storage nodes in read and write operations makes it even more challenging, since the performance of these operations is limited by the slowest of the R or W replicas. Figure 4 shows the average and 99.9th percentile latencies of Dynamo's read and write operations during a period of 30 days. As seen in the figure, the latencies exhibit a clear diurnal pattern which is a result of the diurnal pattern in the incoming request rate (i.e., there is a significant difference in request rate between the daytime and night). Moreover, the write latencies are higher than read latencies obviously because write operations always results in disk access. Also, the 99.9th percentile latencies are around 200 ms and are an order of magnitude higher than the averages. This is because the 99.9th percentile latencies are affected by several factors such as variability in request load, object sizes, and locality patterns.

While this level of performance is acceptable for a number of services, a few customer-facing services required higher levels of performance. For these services, Dynamo provides the ability to trade-off durability guarantees for performance. In the optimization each storage node maintains an object buffer in its main memory. Each write operation is stored in the buffer and gets periodically written to storage by a *writer thread*. In this scheme, read operations first check if the requested key is present in the buffer. If so, the object is read from the buffer instead of the storage engine.

This optimization has resulted in lowering the 99.9th percentile latency by a factor of 5 during peak traffic even for a very small buffer of a thousand objects (see Figure 5). Also, as seen in the figure, write buffering smoothes out higher percentile latencies. Obviously, this scheme trades durability for performance. In this scheme, a server crash can result in missing writes that were queued up in the buffer. To reduce the durability risk, the write operation is refined to have the coordinator choose one out of the N replicas to perform a "durable write". Since the coordinator waits only for W responses, the performance of the write operation is not affected by the performance of the durable write operation performed by a single replica.

6.2 Ensuring Uniform Load distribution

Dynamo uses consistent hashing to partition its key space across its replicas and to ensure uniform load distribution. A uniform key distribution can help us achieve uniform load distribution assuming the access distribution of keys is not highly skewed. In particular, Dynamo's design assumes that even where there is a significant skew in the access distribution there are enough keys in the popular end of the distribution so that the load of handling popular keys can be spread across the nodes uniformly through partitioning. This section discusses the load imbalance seen in Dynamo and the impact of different partitioning strategies on load distribution.

To study the load imbalance and its correlation with request load, the total number of requests received by each node was measured for a period of 24 hours - broken down into intervals of 30 minutes. In a given time window, a node is considered to be "in-balance", if the node's request load deviates from the average load by a value a less than a certain threshold (here 15%). Otherwise the node was deemed "out-of-balance". Figure 6 presents the fraction of nodes that are "out-of-balance" (henceforth, "imbalance ratio") during this time period. For reference, the corresponding request load received by the entire system during this time period is also plotted. As seen in the figure, the imbalance ratio decreases with increasing load. For instance, during low loads the imbalance ratio is as high as 20% and during high loads it is close to 10%. Intuitively, this can be explained by the fact that under high loads, a large number of popular keys are accessed and due to uniform distribution of keys the load is evenly distributed. However, during low loads (where load is 1/8th

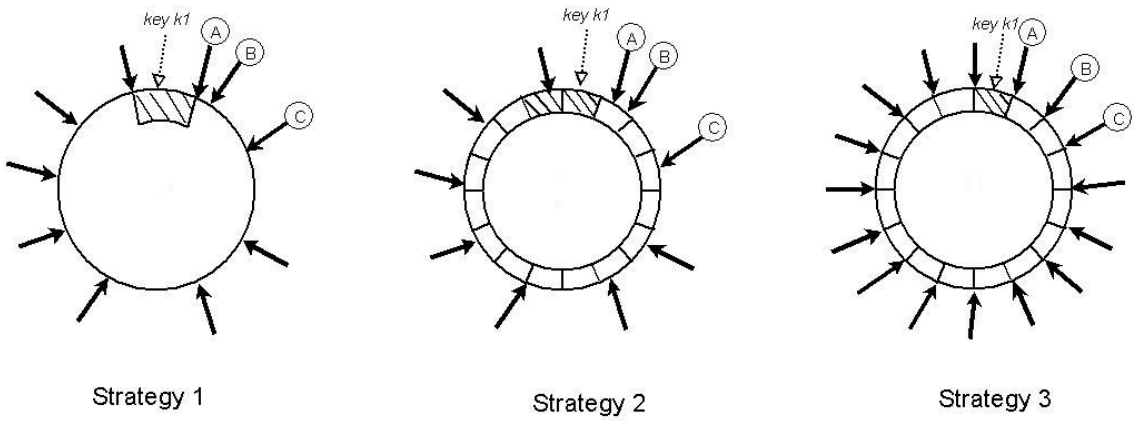


Figure 7: Partitioning and placement of keys in the three strategies. A, B, and C depict the three unique nodes that form the preference list for the key k1 on the consistent hashing ring (N=3). The shaded area indicates the key range for which nodes A, B, and C form the preference list. Dark arrows indicate the token locations for various nodes.

of the measured peak load), fewer popular keys are accessed, resulting in a higher load imbalance.

This section discusses how Dynamo's partitioning scheme has evolved over time and its implications on load distribution.

Strategy 1: T random tokens per node and partition by token value: This was the initial strategy deployed in production (and described in Section 4.2). In this scheme, each node is assigned T tokens (chosen uniformly at random from the hash space). The tokens of all nodes are ordered according to their values in the hash space. Every two consecutive tokens define a range. The last token and the first token form a range that "wraps" around from the highest value to the lowest value in the hash space. Because the tokens are chosen randomly, the ranges vary in size. As nodes join and leave the system, the token set changes and consequently the ranges change. Note that the space needed to maintain the membership at each node increases linearly with the number of nodes in the system.

While using this strategy, the following problems were encountered. First, when a new node joins the system, it needs to "steal" its key ranges from other nodes. However, the nodes handing the key ranges off to the new node have to scan their local persistence store to retrieve the appropriate set of data items. Note that performing such a scan operation on a production node is tricky as scans are highly resource intensive operations and they need to be executed in the background without affecting the customer performance. This requires us to run the bootstrapping task at the lowest priority. However, this significantly slows the bootstrapping process and during busy shopping season, when the nodes are handling millions of requests a day, the bootstrapping has taken almost a day to complete. Second, when a node joins/leaves the system, the key ranges handled by many nodes change and the Merkle trees for the new ranges need to be recalculated, which is a non-trivial operation to perform on a production system. Finally, there was no easy way to take a snapshot of the entire key space due to the randomness in key ranges, and this made the process of archival complicated. In this scheme, archiving the entire key space requires us to retrieve the keys from each node separately, which is highly inefficient.

The fundamental issue with this strategy is that the schemes for data partitioning and data placement are intertwined. For instance, in some cases, it is preferred to add more nodes to the system in order to handle an increase in request load. However, in this scenario, it is not possible to add nodes without affecting data partitioning. Ideally, it is desirable to use independent schemes for partitioning and placement. To this end, following strategies were evaluated:

Strategy 2: T random tokens per node and equal sized partitions: In this strategy, the hash space is divided into Q equally sized partitions/ranges and each node is assigned T random tokens. Q is usually set such that Q >> N and Q >> S*T, where S is the number of nodes in the system. In this strategy, the tokens are only used to build the function that maps values in the hash space to the ordered lists of nodes and not to decide the partitioning. A partition is placed on the first N unique nodes that are encountered while walking the consistent hashing ring clockwise from the end of the partition. Figure 7 illustrates this strategy for N=3. In this example, nodes A, B, C are encountered while walking the ring from the end of the partition that contains key k1. The primary advantages of this strategy are: (i) decoupling of partitioning and partition placement, and (ii) enabling the possibility of changing the placement scheme at runtime.

Strategy 3: Q/S tokens per node, equal-sized partitions: Similar to strategy 2, this strategy divides the hash space into Q equally sized partitions and the placement of partition is decoupled from the partitioning scheme. Moreover, each node is assigned Q/S tokens where S is the number of nodes in the system. When a node leaves the system, its tokens are randomly distributed to the remaining nodes such that these properties are preserved. Similarly, when a node joins the system it "steals" tokens from nodes in the system in a way that preserves these properties.

The efficiency of these three strategies is evaluated for a system with S=30 and N=3. However, comparing these different strategies in a fair manner is hard as different strategies have different configurations to tune their efficiency. For instance, the load distribution property of strategy 1 depends on the number of tokens (i.e., T) while strategy 3 depends on the number of partitions (i.e., Q). One fair way to compare these strategies is to

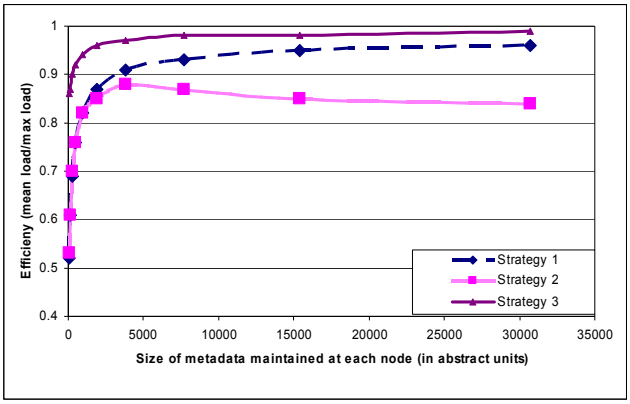


Figure 8: Comparison of the load distribution efficiency of different strategies for system with 30 nodes and N=3 with equal amount of metadata maintained at each node. The values of the system size and number of replicas are based on the typical configuration deployed for majority of our services.

evaluate the skew in their load distribution while all strategies use the same amount of space to maintain their membership information. For instance, in strategy 1 each node needs to maintain the token positions of all the nodes in the ring and in strategy 3 each node needs to maintain the information regarding the partitions assigned to each node.

In our next experiment, these strategies were evaluated by varying the relevant parameters (T and Q). The load balancing efficiency of each strategy was measured for different sizes of membership information that needs to be maintained at each node, where *Load balancing efficiency* is defined as the ratio of average number of requests served by each node to the maximum number of requests served by the hottest node.

The results are given in Figure 8. As seen in the figure, strategy 3 achieves the best load balancing efficiency and strategy 2 has the worst load balancing efficiency. For a brief time, Strategy 2 served as an interim setup during the process of migrating Dynamo instances from using Strategy 1 to Strategy 3. Compared to Strategy 1, Strategy 3 achieves better efficiency and reduces the size of membership information maintained at each node by three orders of magnitude. While storage is not a major issue the nodes gossip the membership information periodically and as such it is desirable to keep this information as compact as possible. In addition to this, strategy 3 is advantageous and simpler to deploy for the following reasons: (i) *Faster bootstrapping/recovery:* Since partition ranges are fixed, they can be stored in separate files, meaning a partition can be relocated as a unit by simply transferring the file (avoiding random accesses needed to locate specific items). This simplifies the process of bootstrapping and recovery. (ii) *Ease of archival*: Periodical archiving of the dataset is a mandatory requirement for most of Amazon storage services. Archiving the entire dataset stored by Dynamo is simpler in strategy 3 because the partition files can be archived separately. By contrast, in Strategy 1, the tokens are chosen randomly and, archiving the data stored in Dynamo requires retrieving the keys from individual nodes separately and is usually inefficient and slow. The disadvantage of strategy 3 is that changing the node membership requires coordination in order to preserve the properties required of the assignment.

6.3 Divergent Versions: When and How Many?

As noted earlier, Dynamo is designed to tradeoff consistency for availability. To understand the precise impact of different failures on consistency, detailed data is required on multiple factors: outage length, type of failure, component reliability, workload etc. Presenting these numbers in detail is outside of the scope of this paper. However, this section discusses a good summary metric: the number of divergent versions seen by the application in a live production environment.

Divergent versions of a data item arise in two scenarios. The first is when the system is facing failure scenarios such as node failures, data center failures, and network partitions. The second is when the system is handling a large number of concurrent writers to a single data item and multiple nodes end up coordinating the updates concurrently. From both a usability and efficiency perspective, it is preferred to keep the number of divergent versions at any given time as low as possible. If the versions cannot be syntactically reconciled based on vector clocks alone, they have to be passed to the business logic for semantic reconciliation. Semantic reconciliation introduces additional load on services, so it is desirable to minimize the need for it.

In our next experiment, the number of versions returned to the shopping cart service was profiled for a period of 24 hours. During this period, 99.94% of requests saw exactly one version; 0.00057% of requests saw 2 versions; 0.00047% of requests saw 3 versions and 0.00009% of requests saw 4 versions. This shows that divergent versions are created rarely.

Experience shows that the increase in the number of divergent versions is contributed not by failures but due to the increase in number of concurrent writers. The increase in the number of concurrent writes is usually triggered by busy robots (automated client programs) and rarely by humans. This issue is not discussed in detail due to the sensitive nature of the story.

6.4 Client-driven or Server-driven Coordination

As mentioned in Section 5, Dynamo has a request coordination component that uses a state machine to handle incoming requests. Client requests are uniformly assigned to nodes in the ring by a load balancer. Any Dynamo node can act as a coordinator for a read request. Write requests on the other hand will be coordinated by a node in the key's current preference list. This restriction is due to the fact that these preferred nodes have the added responsibility of creating a new version stamp that causally subsumes the version that has been updated by the write request. Note that if Dynamo's versioning scheme is based on physical timestamps, any node can coordinate a write request.

An alternative approach to request coordination is to move the state machine to the client nodes. In this scheme client applications use a library to perform request coordination locally. A client periodically picks a random Dynamo node and downloads its current view of Dynamo membership state. Using this information the client can determine which set of nodes form the preference list for any given key. Read requests can be coordinated at the client node thereby avoiding the extra network hop that is incurred if the request were assigned to a random Dynamo node by the load balancer. Writes will either be forwarded to a node in the key's preference list or can be

Table 2: Performance of client-driven and server-driven coordination approaches.

	99.9th percentile read latency (ms)	99.9th percentile write latency (ms)	Average read latency (ms)	Average write latency (ms)
Server-driven	68.9	68.5	3.9	4.02
Client-driven	30.4	30.4	1.55	1.9

coordinated locally if Dynamo is using timestamps based versioning.

An important advantage of the client-driven coordination approach is that a load balancer is no longer required to uniformly distribute client load. Fair load distribution is implicitly guaranteed by the near uniform assignment of keys to the storage nodes. Obviously, the efficiency of this scheme is dependent on how fresh the membership information is at the client. Currently clients poll a random Dynamo node every 10 seconds for membership updates. A pull based approach was chosen over a push based one as the former scales better with large number of clients and requires very little state to be maintained at servers regarding clients. However, in the worst case the client can be exposed to stale membership for duration of 10 seconds. In case, if the client detects its membership table is stale (for instance, when some members are unreachable), it will immediately refresh its membership information.

Table 2 shows the latency improvements at the 99.9th percentile and averages that were observed for a period of 24 hours using client-driven coordination compared to the server-driven approach. As seen in the table, the client-driven coordination approach reduces the latencies by at least 30 milliseconds for 99.9th percentile latencies and decreases the average by 3 to 4 milliseconds. The latency improvement is because the client-driven approach eliminates the overhead of the load balancer and the extra network hop that may be incurred when a request is assigned to a random node. As seen in the table, average latencies tend to be significantly lower than latencies at the 99.9th percentile. This is because Dynamo's storage engine caches and write buffer have good hit ratios. Moreover, since the load balancers and network introduce additional variability to the response time, the gain in response time is higher for the 99.9th percentile than the average.

6.5 Balancing background vs. foreground tasks

Each node performs different kinds of background tasks for replica synchronization and data handoff (either due to hinting or adding/removing nodes) in addition to its normal foreground put/get operations. In early production settings, these background tasks triggered the problem of resource contention and affected the performance of the regular put and get operations. Hence, it became necessary to ensure that background tasks ran only when the regular critical operations are not affected significantly. To this end, the background tasks were integrated with an admission control mechanism. Each of the background tasks uses this controller to reserve runtime slices of the resource (e.g. database), shared across all background tasks. A feedback mechanism based on the monitored performance of the foreground tasks is employed to change the number of slices that are available to the background tasks.

The admission controller constantly monitors the behavior of resource accesses while executing a "foreground" put/get operation. Monitored aspects include latencies for disk operations, failed database accesses due to lock-contention and transaction timeouts, and request queue wait times. This information is used to check whether the percentiles of latencies (or failures) in a given trailing time window are close to a desired threshold. For example, the background controller checks to see how close the 99th percentile database read latency (over the last 60 seconds) is to a preset threshold (say 50ms). The controller uses such comparisons to assess the resource availability for the foreground operations. Subsequently, it decides on how many time slices will be available to background tasks, thereby using the feedback loop to limit the intrusiveness of the background activities. Note that a similar problem of managing background tasks has been studied in [4].

6.6 Discussion

This section summarizes some of the experiences gained during the process of implementation and maintenance of Dynamo. Many Amazon internal services have used Dynamo for the past two years and it has provided significant levels of availability to its applications. In particular, applications have received successful responses (without timing out) for 99.9995% of its requests and no data loss event has occurred to date.

Moreover, the primary advantage of Dynamo is that it provides the necessary knobs using the three parameters of (N,R,W) to tune their instance based on their needs.. Unlike popular commercial data stores, Dynamo exposes data consistency and reconciliation logic issues to the developers. At the outset, one may expect the application logic to become more complex. However, historically, Amazon's platform is built for high availability and many applications are designed to handle different failure modes and inconsistencies that may arise. Hence, porting such applications to use Dynamo was a relatively simple task. For new applications that want to use Dynamo, some analysis is required during the initial stages of the development to pick the right conflict resolution mechanisms that meet the business case appropriately. Finally, Dynamo adopts a full membership model where each node is aware of the data hosted by its peers. To do this, each node actively gossips the full routing table with other nodes in the system. This model works well for a system that contains couple of hundreds of nodes. However, scaling such a design to run with tens of thousands of nodes is not trivial because the overhead in maintaining the routing table increases with the system size. This limitation might be overcome by introducing hierarchical extensions to Dynamo. Also, note that this problem is actively addressed by O(1) DHT systems(e.g., [14]).

7. CONCLUSIONS

This paper described Dynamo, a highly available and scalable data store, used for storing state of a number of core services of Amazon.com's e-commerce platform. Dynamo has provided the desired levels of availability and performance and has been successful in handling server failures, data center failures and network partitions. Dynamo is incrementally scalable and allows service owners to scale up and down based on their current

request load. Dynamo allows service owners to customize their storage system to meet their desired performance, durability and consistency SLAs by allowing them to tune the parameters N, R, and W.

The production use of Dynamo for the past year demonstrates that decentralized techniques can be combined to provide a single highly-available system. Its success in one of the most challenging application environments shows that an eventual-consistent storage system can be a building block for highly-available applications.

ACKNOWLEDGEMENTS

The authors would like to thank Pat Helland for his contribution to the initial design of Dynamo. We would also like to thank Marvin Theimer and Robert van Renesse for their comments. Finally, we would like to thank our shepherd, Jeff Mogul, for his detailed comments and inputs while preparing the camera ready version that vastly improved the quality of the paper.

REFERENCES

[1] Adya, A., Bolosky, W. J., Castro, M., Cermak, G., Chaiken, R., Douceur, J. R., Howell, J., Lorch, J. R., Theimer, M., and Wattenhofer, R. P. 2002. Farsite: federated, available, and reliable storage for an incompletely trusted environment. *SIGOPS Oper. Syst. Rev.* 36, SI (Dec. 2002), 1-14.

[2] Bernstein, P.A., and Goodman, N. An algorithm for concurrency control and recovery in replicated distributed databases. ACM Trans. on Database Systems, 9(4):596-615, December 1984

[3] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E. 2006. Bigtable: a distributed storage system for structured data. In *Proceedings of the 7th Conference on USENIX Symposium on Operating Systems Design and Implementation - Volume 7* (Seattle, WA, November 06 - 08, 2006). USENIX Association, Berkeley, CA, 15-15.

[4] Douceur, J. R. and Bolosky, W. J. 2000. Process-based regulation of low-importance processes. *SIGOPS Oper. Syst. Rev.* 34, 2 (Apr. 2000), 26-27.

[5] Fox, A., Gribble, S. D., Chawathe, Y., Brewer, E. A., and Gauthier, P. 1997. Cluster-based scalable network services. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles* (Saint Malo, France, October 05 - 08, 1997). W. M. Waite, Ed. SOSP '97. ACM Press, New York, NY, 78-91.

[6] Ghemawat, S., Gobioff, H., and Leung, S. 2003. The Google file system. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles* (Bolton Landing, NY, USA, October 19 - 22, 2003). SOSP '03. ACM Press, New York, NY, 29-43.

[7] Gray, J., Helland, P., O'Neil, P., and Shasha, D. 1996. The dangers of replication and a solution. In *Proceedings of the 1996 ACM SIGMOD international Conference on Management of Data* (Montreal, Quebec, Canada, June 04 - 06, 1996). J. Widom, Ed. SIGMOD '96. ACM Press, New York, NY, 173-182.

[8] Gupta, I., Chandra, T. D., and Goldszmidt, G. S. 2001. On scalable and efficient distributed failure detectors. In *Proceedings of the Twentieth Annual ACM Symposium on Principles of Distributed Computing* (Newport, Rhode Island, United States). PODC '01. ACM Press, New York, NY, 170-179.

[9] Kubiatowicz, J., Bindel, D., Chen, Y., Czerwinski, S., Eaton, P., Geels, D., Gummadi, R., Rhea, S., Weatherspoon, H., Wells, C., and Zhao, B. 2000. OceanStore: an architecture for global-scale persistent storage. *SIGARCH Comput. Archit. News* 28, 5 (Dec. 2000), 190-201.

[10] Karger, D., Lehman, E., Leighton, T., Panigrahy, R., Levine, M., and Lewin, D. 1997. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web. In *Proceedings of the Twenty-Ninth Annual ACM Symposium on theory of Computing* (El Paso, Texas, United States, May 04 - 06, 1997). STOC '97. ACM Press, New York, NY, 654-663.

[11] Lindsay, B.G., et. al., "Notes on Distributed Databases", Research Report RJ2571(33471), IBM Research, July 1979

[12] Lamport, L. Time, clocks, and the ordering of events in a distributed system. ACM Communications, 21(7), pp. 558-565, 1978.

[13] Merkle, R. A digital signature based on a conventional encryption function. Proceedings of CRYPTO, pages 369–378. Springer-Verlag, 1988.

[14] Ramasubramanian, V., and Sirer, E. G. Beehive: O(1)lookup performance for power-law query distributions in peer-to-peer overlays. In *Proceedings of the 1st Conference on Symposium on Networked Systems Design and Implementation,* San Francisco, CA, March 29 - 31, 2004.

[15] Reiher, P., Heidemann, J., Ratner, D., Skinner, G., and Popek, G. 1994. Resolving file conflicts in the Ficus file system. In *Proceedings of the USENIX Summer 1994 Technical Conference on USENIX Summer 1994 Technical Conference - Volume 1* (Boston, Massachusetts, June 06 - 10, 1994). USENIX Association, Berkeley, CA, 12-12..

[16] Rowstron, A., and Druschel, P. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. Proceedings of Middleware, pages 329-350, November, 2001.

[17] Rowstron, A., and Druschel, P. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. Proceedings of Symposium on Operating Systems Principles, October 2001.

[18] Saito, Y., Frølund, S., Veitch, A., Merchant, A., and Spence, S. 2004. FAB: building distributed enterprise disk arrays from commodity components. *SIGOPS Oper. Syst. Rev.* 38, 5 (Dec. 2004), 48-58.

[19] Satyanarayanan, M., Kistler, J.J., Siegel, E.H. Coda: A Resilient Distributed File System. IEEE Workshop on Workstation Operating Systems, Nov. 1987.

[20] Stoica, I., Morris, R., Karger, D., Kaashoek, M. F., and Balakrishnan, H. 2001. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols For Computer Communications* (San Diego, California, United States). SIGCOMM '01. ACM Press, New York, NY, 149-160.

[21] Terry, D. B., Theimer, M. M., Petersen, K., Demers, A. J., Spreitzer, M. J., and Hauser, C. H. 1995. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles* (Copper Mountain, Colorado, United States, December 03 - 06, 1995). M. B. Jones, Ed. SOSP '95. ACM Press, New York, NY, 172-182.

[22] Thomas, R. H. A majority consensus approach to concurrency control for multiple copy databases. ACM Transactions on Database Systems 4 (2): 180-209, 1979.

[23] Weatherspoon, H., Eaton, P., Chun, B., and Kubiatowicz, J. 2007. Antiquity: exploiting a secure log for wide-area distributed storage. *SIGOPS Oper. Syst. Rev.* 41, 3 (Jun. 2007), 371-384.

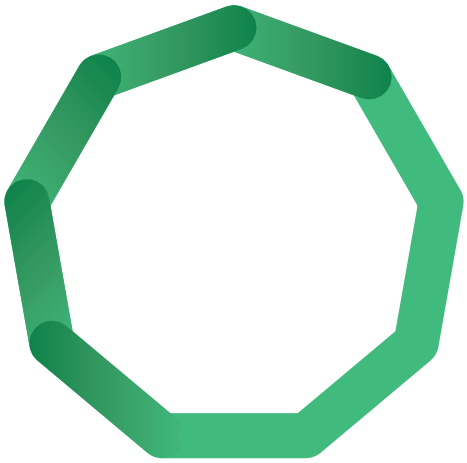
[24] Welsh, M., Culler, D., and Brewer, E. 2001. SEDA: an architecture for well-conditioned, scalable internet services. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles* (Banff, Alberta, Canada, October 21 - 24, 2001). SOSP '01. ACM Press, New York, NY, 230-243.

Chapter 11

The Chubby lock service for loosely-coupled distributed systems

Authors:
Mike Burrows

Publication year: 2006



The Chubby lock service for loosely-coupled distributed systems

Mike Burrows, *Google Inc.*

Abstract

We describe our experiences with the Chubby lock service, which is intended to provide coarse-grained locking as well as reliable (though low-volume) storage for a loosely-coupled distributed system. Chubby provides an interface much like a distributed file system with advisory locks, but the design emphasis is on availability and reliability, as opposed to high performance. Many instances of the service have been used for over a year, with several of them each handling a few tens of thousands of clients concurrently. The paper describes the initial design and expected use, compares it with actual use, and explains how the design had to be modified to accommodate the differences.

1 Introduction

This paper describes a *lock service* called Chubby. It is intended for use within a loosely-coupled distributed system consisting of moderately large numbers of small machines connected by a high-speed network. For example, a Chubby instance (also known as a Chubby *cell*) might serve ten thousand 4-processor machines connected by 1Gbit/s Ethernet. Most Chubby cells are confined to a single data centre or machine room, though we do run at least one Chubby cell whose replicas are separated by thousands of kilometres.

The purpose of the lock service is to allow its clients to synchronize their activities and to agree on basic information about their environment. The primary goals included reliability, availability to a moderately large set of clients, and easy-to-understand semantics; throughput and storage capacity were considered secondary. Chubby's client interface is similar to that of a simple file system that performs *whole-file* reads and writes, augmented with advisory locks and with notification of various events such as file modification.

We expected Chubby to help developers deal with coarse-grained synchronization within their systems, and in particular to deal with the problem of electing a leader from among a set of otherwise equivalent servers. For example, the Google File System [7] uses a Chubby lock to appoint a GFS master server, and Bigtable [3] uses Chubby in several ways: to elect a master, to allow the master to discover the servers it controls, and to permit clients to find the master. In addition, both GFS and Bigtable use Chubby as a well-known and available location to store a small amount of meta-data; in effect they use Chubby as the root of their distributed data structures. Some services use locks to partition work (at a coarse grain) between several servers.

Before Chubby was deployed, most distributed systems at Google used *ad hoc* methods for primary election (when work could be duplicated without harm), or required operator intervention (when correctness was essential). In the former case, Chubby allowed a small saving in computing effort. In the latter case, it achieved a significant improvement in availability in systems that no longer required human intervention on failure.

Readers familiar with distributed computing will recognize the election of a primary among peers as an instance of the *distributed consensus* problem, and realize we require a solution using *asynchronous* communication; this term describes the behaviour of the vast majority of real networks, such as Ethernet or the Internet, which allow packets to be lost, delayed, and reordered. (Practitioners should normally beware of protocols based on models that make stronger assumptions on the environment.) Asynchronous consensus is solved by the *Paxos* protocol [12, 13]. The same protocol was used by Oki and Liskov (see their paper on *viewstamped replication* [19, §4]), an equivalence noted by others [14, §6]. Indeed, all working protocols for asynchronous consensus we have so far encountered have Paxos at their core. Paxos maintains safety without timing assumptions, but clocks must be introduced to ensure liveness; this overcomes the impossibility result of Fischer *et al.* [5, §1].

Building Chubby was an engineering effort required to fill the needs mentioned above; it was not research. We claim no new algorithms or techniques. The purpose of this paper is to describe what we did and why, rather than to advocate it. In the sections that follow, we describe Chubby's design and implementation, and how it

has changed in the light of experience. We describe unexpected ways in which Chubby has been used, and features that proved to be mistakes. We omit details that are covered elsewhere in the literature, such as the details of a consensus protocol or an RPC system.

2 Design

2.1 Rationale

One might argue that we should have built a library embodying Paxos, rather than a library that accesses a centralized lock service, even a highly reliable one. A client Paxos library would depend on *no* other servers (besides the name service), and would provide a standard framework for programmers, assuming their services can be implemented as state machines. Indeed, we provide such a client library that is independent of Chubby.

Nevertheless, a lock service has some advantages over a client library. First, our developers sometimes do not plan for high availability in the way one would wish. Often their systems start as prototypes with little load and loose availability guarantees; invariably the code has not been specially structured for use with a consensus protocol. As the service matures and gains clients, availability becomes more important; replication and primary election are then added to an existing design. While this could be done with a library that provides distributed consensus, a lock server makes it easier to maintain existing program structure and communication patterns. For example, to elect a master which then writes to an existing file server requires adding just two statements and one RPC parameter to an existing system: One would acquire a lock to become master, pass an additional integer (the lock acquisition count) with the write RPC, and add an if-statement to the file server to reject the write if the acquisition count is lower than the current value (to guard against delayed packets). We have found this technique easier than making existing servers participate in a consensus protocol, and especially so if compatibility must be maintained during a transition period.

Second, many of our services that elect a primary or that partition data between their components need a mechanism for advertising the results. This suggests that we should allow clients to store and fetch small quantities of data—that is, to read and write small files. This could be done with a name service, but our experience has been that the lock service itself is well-suited for this task, both because this reduces the number of servers on which a client depends, and because the consistency features of the protocol are shared. Chubby's success as a name server owes much to its use of consistent client caching, rather than time-based caching. In particular, we found that developers greatly appreciated not having to choose a cache timeout such as the DNS time-to-live value, which if chosen poorly can lead to high DNS load, or long client fail-over times.

Third, a lock-based interface is more familiar to our programmers. Both the replicated state machine of Paxos and the critical sections associated with exclusive locks can provide the programmer with the illusion of sequential programming. However, many programmers have come across locks before, and think they know to use them. Ironically, such programmers are usually wrong, especially when they use locks in a distributed system; few consider the effects of independent machine failures on locks in a system with asynchronous communications. Nevertheless, the apparent familiarity of locks overcomes a hurdle in persuading programmers to use a reliable mechanism for distributed decision making.

Last, distributed-consensus algorithms use quorums to make decisions, so they use several replicas to achieve high availability. For example, Chubby itself usually has five replicas in each cell, of which three must be running for the cell to be up. In contrast, if a client system uses a lock service, even a single client can obtain a lock and make progress safely. Thus, a lock service reduces the number of servers needed for a reliable client system to make progress. In a loose sense, one can view the lock service as a way of providing a generic electorate that allows a client system to make decisions correctly when less than a majority of its own members are up. One might imagine solving this last problem in a different way: by providing a "consensus service", using a number of servers to provide the "acceptors" in the Paxos protocol. Like a lock service, a consensus service would allow clients to make progress safely even with only one active client process; a similar technique has been used to reduce the number of state machines needed for Byzantine fault tolerance [24]. However, assuming a consensus service is not used exclusively to provide locks (which reduces it to a lock service), this approach solves none of the other problems described above.

These arguments suggest two key design decisions:

- We chose a lock service, as opposed to a library or service for consensus, and
- we chose to serve small-files to permit elected primaries to advertise themselves and their parameters, rather than build and maintain a second service.

Some decisions follow from our expected use and from our environment:

- A service advertising its primary via a Chubby file may have thousands of clients. Therefore, we must allow thousands of clients to observe this file, preferably without needing many servers.
- Clients and replicas of a replicated service may wish to know when the service's primary changes. This

suggests that an event notification mechanism would be useful to avoid polling.

- Even if clients need not poll files periodically, many will; this is a consequence of supporting many developers. Thus, caching of files is desirable.
- Our developers are confused by non-intuitive caching semantics, so we prefer consistent caching.
- To avoid both financial loss and jail time, we provide security mechanisms, including access control.

A choice that may surprise some readers is that we do not expect lock use to be *fine-grained*, in which they might be held only for a short duration (seconds or less); instead, we expect *coarse-grained* use. For example, an application might use a lock to elect a primary, which would then handle all access to that data for a considerable time, perhaps hours or days. These two styles of use suggest different requirements from a lock server.

Coarse-grained locks impose far less load on the lock server. In particular, the lock-acquisition rate is usually only weakly related to the transaction rate of the client applications. Coarse-grained locks are acquired only rarely, so temporary lock server unavailability delays clients less. On the other hand, the transfer of a lock from client to client may require costly recovery procedures, so one would not wish a fail-over of a lock server to cause locks to be lost. Thus, it is good for coarse-grained locks to survive lock server failures, there is little concern about the overhead of doing so, and such locks allow many clients to be adequately served by a modest number of lock servers with somewhat lower availability.

Fine-grained locks lead to different conclusions. Even brief unavailability of the lock server may cause many clients to stall. Performance and the ability to add new servers at will are of great concern because the transaction rate at the lock service grows with the combined transaction rate of clients. It can be advantageous to reduce the overhead of locking by not maintaining locks across lock server failure, and the time penalty for dropping locks every so often is not severe because locks are held for short periods. (Clients must be prepared to lose locks during network partitions, so the loss of locks on lock server fail-over introduces no new recovery paths.)

Chubby is intended to provide only coarse-grained locking. Fortunately, it is straightforward for clients to implement their own fine-grained locks tailored to their application. An application might partition its locks into groups and use Chubby's coarse-grained locks to allocate these lock groups to application-specific lock servers. Little state is needed to maintain these fine-grain locks; the servers need only keep a non-volatile, monotonically-increasing acquisition counter that is rarely updated. Clients can learn of lost locks at unlock time, and if a simple fixed-length lease is used, the protocol can be simple and efficient. The most important benefits of this scheme are that our client developers become responsible for the provisioning of the servers needed to support their load, yet are relieved of the complexity of implementing consensus themselves.

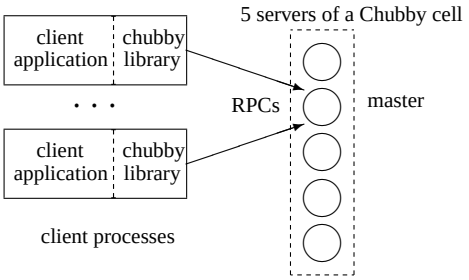


Figure 1: System structure

2.2 System structure

Chubby has two main components that communicate via RPC: a server, and a library that client applications link against; see Figure 1. All communication between Chubby clients and the servers is mediated by the client library. An optional third component, a proxy server, is discussed in Section 3.1.

A Chubby cell consists of a small set of servers (typically five) known as *replicas*, placed so as to reduce the likelihood of correlated failure (for example, in different racks). The replicas use a distributed consensus protocol to elect a *master*; the master must obtain votes from a majority of the replicas, plus promises that those replicas will not elect a different master for an interval of a few seconds known as the *master lease*. The master lease is periodically renewed by the replicas provided the master continues to win a majority of the vote.

The replicas maintain copies of a simple database, but only the master initiates reads and writes of this database. All other replicas simply copy updates from the master, sent using the consensus protocol.

Clients find the master by sending master location requests to the replicas listed in the DNS. Non-master replicas respond to such requests by returning the identity of the master. Once a client has located the master, the client directs all requests to it either until it ceases to respond, or until it indicates that it is no longer the master. Write requests are propagated via the consensus protocol to all replicas; such requests are acknowledged when the write has reached a majority of the replicas in the cell. Read requests are satisfied by the master alone; this is safe provided the master lease has not expired, as no other master can possibly exist. If a master fails, the other replicas run the election protocol when their master leases expire; a new master will typically be elected in a few seconds. For example, two recent elections took 6s and 4s, but we see values as high as 30s (§4.1).

If a replica fails and does not recover for a few hours, a simple replacement system selects a fresh machine from a free pool and starts the lock server binary on it. It then updates the DNS tables, replacing the IP address of the failed replica with that of the new one. The current master polls the DNS periodically and eventually notices the change. It then updates the list of the cell's members in the cell's database; this list is kept consistent across all the members via the normal replication protocol. In the meantime, the new replica obtains a recent copy of the database from a combination of backups stored on file servers and updates from active replicas. Once the new replica has processed a request that the current master is waiting to commit, the replica is permitted to vote in the elections for new master.

2.3 Files, directories, and handles

Chubby exports a file system interface similar to, but simpler than that of UNIX [22]. It consists of a strict tree of files and directories in the usual way, with name components separated by slashes. A typical name is:

`/ls/foo/wombat/pouch`

The `ls` prefix is common to all Chubby names, and stands for *lock service*. The second component (`foo`) is the name of a Chubby cell; it is resolved to one or more Chubby servers via DNS lookup. A special cell name `local` indicates that the client's local Chubby cell should be used; this is usually one in the same building and thus the one most likely to be accessible. The remainder of the name, `/wombat/pouch`, is interpreted within the named Chubby cell. Again following UNIX, each directory contains a list of child files and directories, while each file contains a sequence of uninterpreted bytes.

Because Chubby's naming structure resembles a file system, we were able to make it available to applications both with its own specialized API, and via interfaces used by our other file systems, such as the Google File System. This significantly reduced the effort needed to write basic browsing and name space manipulation tools, and reduced the need to educate casual Chubby users.

The design differs from UNIX in a ways that ease distribution. To allow the files in different directories to be served from different Chubby masters, we do not expose operations that can move files from one directory to another, we do not maintain directory modified times, and we avoid path-dependent permission semantics (that is, access to a file is controlled by the permissions on the file itself rather than on directories on the path leading to the file). To make it easier to cache file meta-data, the system does not reveal last-access times.

The name space contains only files and directories, collectively called *nodes*. Every such node has only one name within its cell; there are no symbolic or hard links.

Nodes may be either permanent or ephemeral. Any node may be deleted explicitly, but ephemeral nodes are also deleted if no client has them open (and, for directories, they are empty). Ephemeral files are used as temporary files, and as indicators to others that a client is alive. Any node can act as an advisory reader/writer lock; these locks are described in more detail in Section 2.4.

Each node has various meta-data, including three names of access control lists (ACLs) used to control reading, writing and changing the ACL names for the node. Unless overridden, a node inherits the ACL names of its parent directory on creation. ACLs are themselves files located in an ACL directory, which is a well-known part of the cell's local name space. These ACL files consist of simple lists of names of principals; readers may be reminded of Plan 9's *groups* [21]. Thus, if file F's write ACL name is `foo`, and the ACL directory contains a file `foo` that contains an entry `bar`, then user `bar` is permitted to write F. Users are authenticated by a mechanism built into the RPC system. Because Chubby's ACLs are simply files, they are automatically available to other services that wish to use similar access control mechanisms.

The per-node meta-data includes four monotonically-increasing 64-bit numbers that allow clients to detect changes easily:

- an instance number; greater than the instance number of any previous node with the same name.
- a content generation number (files only); this increases when the file's contents are written.
- a lock generation number; this increases when the node's lock transitions from *free* to *held*.
- an ACL generation number; this increases when the node's ACL names are written.

Chubby also exposes a 64-bit file-content checksum so clients may tell whether files differ.

Clients open nodes to obtain *handles* that are analogous to UNIX file descriptors. Handles include:

- check digits that prevent clients from creating or guessing handles, so full access control checks need be performed only when handles are created (compare with UNIX, which checks its permissions bits at open time, but not at each read/write because file descriptors cannot be forged).
- a sequence number that allows a master to tell whether a handle was generated by it or by a previous master.
- mode information provided at open time to allow the master to recreate its state if an old handle is presented to a newly restarted master.

2.4 Locks and sequencers

Each Chubby file and directory can act as a reader-writer lock: either one client handle may hold the lock in exclusive (writer) mode, or any number of client handles may

hold the lock in shared (reader) mode. Like the mutexes known to most programmers, locks are *advisory*. That is, they conflict only with other attempts to acquire the same lock: holding a lock called F neither is necessary to access the file F, nor prevents other clients from doing so. We rejected *mandatory* locks, which make locked objects inaccessible to clients not holding their locks:

- Chubby locks often protect resources implemented by other services, rather than just the file associated with the lock. To enforce mandatory locking in a meaningful way would have required us to make more extensive modification of these services.
- We did not wish to force users to shut down applications when they needed to access locked files for debugging or administrative purposes. In a complex system, it is harder to use the approach employed on most personal computers, where administrative software can break mandatory locks simply by instructing the user to shut down his applications or to reboot.
- Our developers perform error checking in the conventional way, by writing assertions such as "lock X is held", so they benefit little from mandatory checks. Buggy or malicious processes have many opportunities to corrupt data when locks are not held, so we find the extra guards provided by mandatory locking to be of no significant value.

In Chubby, acquiring a lock in either mode requires write permission so that an unprivileged reader cannot prevent a writer from making progress.

Locking is complex in distributed systems because communication is typically uncertain, and processes may fail independently. Thus, a process holding a lock *L* may issue a request *R*, but then fail. Another process may acquire *L* and perform some action before *R* arrives at its destination. If *R* later arrives, it may be acted on without the protection of *L*, and potentially on inconsistent data. The problem of receiving messages out of order has been well studied; solutions include *virtual time* [11], and *virtual synchrony* [1], which avoids the problem by ensuring that messages are processed in an order consistent with the observations of every participant.

It is costly to introduce sequence numbers into all the interactions in an existing complex system. Instead, Chubby provides a means by which sequence numbers can be introduced into only those interactions that make use of locks. At any time, a lock holder may request a *sequencer*, an opaque byte-string that describes the state of the lock immediately after acquisition. It contains the name of the lock, the mode in which it was acquired (exclusive or shared), and the lock generation number. The client passes the sequencer to servers (such as file servers) if it expects the operation to be protected by the lock. The recipient server is expected to test whether the sequencer is still valid and has the appropriate mode; if not, it should reject the request. The validity of a sequencer can be checked against the server's Chubby cache or, if the server does not wish to maintain a session with Chubby, against the most recent sequencer that the server has observed. The sequencer mechanism requires only the addition of a string to affected messages, and is easily explained to our developers.

Although we find sequencers simple to use, important protocols evolve slowly. Chubby therefore provides an imperfect but easier mechanism to reduce the risk of delayed or re-ordered requests to servers that do not support sequencers. If a client releases a lock in the normal way, it is immediately available for other clients to claim, as one would expect. However, if a lock becomes free because the holder has failed or become inaccessible, the lock server will prevent other clients from claiming the lock for a period called the *lock-delay*. Clients may specify any lock-delay up to some bound, currently one minute; this limit prevents a faulty client from making a lock (and thus some resource) unavailable for an arbitrarily long time. While imperfect, the lock-delay protects unmodified servers and clients from everyday problems caused by message delays and restarts.

2.5 Events

Chubby clients may subscribe to a range of events when they create a handle. These events are delivered to the client asynchronously via an up-call from the Chubby library. Events include:

- file contents modified—often used to monitor the location of a service advertised via the file.
- child node added, removed, or modified—used to implement mirroring (§2.12). (In addition to allowing new files to be discovered, returning events for child nodes makes it possible to monitor ephemeral files without affecting their reference counts.)
- Chubby master failed over—warns clients that other events may have been lost, so data must be rescanned.
- a handle (and its lock) has become invalid—this typically suggests a communications problem.
- lock acquired—can be used to determine when a primary has been elected.
- conflicting lock request from another client—allows the caching of locks.

Events are delivered after the corresponding action has taken place. Thus, if a client is informed that file contents have changed, it is guaranteed to see the new data (or data that is yet more recent) if it subsequently reads the file.

The last two events mentioned are rarely used, and with hindsight could have been omitted. After primary election for example, clients typically need to communicate with the new primary, rather than simply know that a primary exists; thus, they wait for a file modifi-

cation event indicating that the new primary has written its address in a file. The conflicting lock event in theory permits clients to cache data held on other servers, using Chubby locks to maintain cache consistency. A notification of a conflicting lock request would tell a client to finish using data associated with the lock: it would finish pending operations, flush modifications to a home location, discard cached data, and release. So far, no one has adopted this style of use.

2.6 API

Clients see a Chubby handle as a pointer to an opaque structure that supports various operations. Handles are created only by `Open()`, and destroyed with `Close()`.

`Open()` opens a named file or directory to produce a handle, analogous to a UNIX file descriptor. Only this call takes a node name; all others operate on handles.

The name is evaluated relative to an existing directory handle; the library provides a handle on "/" that is always valid. Directory handles avoid the difficulties of using a program-wide *current directory* in a multi-threaded program that contains many layers of abstraction [18].

The client indicates various options:

- how the handle will be used (reading; writing and locking; changing the ACL); the handle is created only if the client has the appropriate permissions.
- events that should be delivered (see §2.5).
- the lock-delay (§2.4).
- whether a new file or directory should (or must) be created. If a file is created, the caller may supply initial contents and initial ACL names. The return value indicates whether the file was in fact created.

`Close()` closes an open handle. Further use of the handle is not permitted. This call never fails. A related call `Poison()` causes outstanding and subsequent operations on the handle to fail without closing it; this allows a client to cancel Chubby calls made by other threads without fear of deallocating the memory being accessed by them.

The main calls that act on a handle are:

`GetContentsAndStat()` returns both the contents and meta-data of a file. The contents of a file are read atomically and in their entirety. We avoided partial reads and writes to discourage large files. A related call `GetStat()` returns just the meta-data, while `ReadDir()` returns the names and meta-data for the children of a directory.

`SetContents()` writes the contents of a file. Optionally, the client may provide a content generation number to allow the client to simulate compare-and-swap on a file; the contents are changed only if the generation number is current. The contents of a file are always written atomically and in their entirety. A related call `SetACL()` performs a similar operation on the ACL names associated with the node.

`Delete()` deletes the node if it has no children.

`Acquire(), TryAcquire(), Release()` acquire and release locks.

`GetSequencer()` returns a sequencer (§2.4) that describes any lock held by this handle.

`SetSequencer()` associates a sequencer with a handle. Subsequent operations on the handle fail if the sequencer is no longer valid.

`CheckSequencer()` checks whether a sequencer is valid (see §2.4).

Calls fail if the node has been deleted since the handle was created, even if the file has been subsequently recreated. That is, a handle is associated with an instance of a file, rather than with a file name. Chubby may apply access control checks on any call, but always checks `Open()` calls (see §2.3).

All the calls above take an *operation* parameter in addition to any others needed by the call itself. The operation parameter holds data and control information that may be associated with any call. In particular, via the operation parameter the client may:

- supply a callback to make the call asynchronous,
- wait for the completion of such a call, and/or
- obtain extended error and diagnostic information.

Clients can use this API to perform primary election as follows: All potential primaries open the lock file and attempt to acquire the lock. One succeeds and becomes the primary, while the others act as replicas. The primary writes its identity into the lock file with `SetContents()` so that it can be found by clients and replicas, which read the file with `GetContentsAndStat()`, perhaps in response to a file-modification event (§2.5). Ideally, the primary obtains a sequencer with `GetSequencer()`, which it then passes to servers it communicates with; they should confirm with `CheckSequencer()` that it is still the primary. A lock-delay may be used with services that cannot check sequencers (§2.4).

2.7 Caching

To reduce read traffic, Chubby clients cache file data and node meta-data (including file absence) in a consistent, write-through cache held in memory. The cache is maintained by a lease mechanism described below, and kept consistent by invalidations sent by the master, which keeps a list of what each client may be caching. The protocol ensures that clients see either a consistent view of Chubby state, or an error.

When file data or meta-data is to be changed, the modification is blocked while the master sends invalidations for the data to every client that may have cached it; this mechanism sits on top of KeepAlive RPCs, discussed more fully in the next section. On receipt of an invalidation, a client flushes the invalidated state and acknowl-

edges by making its next KeepAlive call. The modification proceeds only after the server knows that each client has invalidated its cache, either because the client acknowledged the invalidation, or because the client allowed its cache lease to expire.

Only one round of invalidations is needed because the master treats the node as *uncachable* while cache invalidations remain unacknowledged. This approach allows reads always to be processed without delay; this is useful because reads greatly outnumber writes. An alternative would be to block calls that access the node during invalidation; this would make it less likely that over-eager clients will bombard the master with uncached accesses during invalidation, at the cost of occasional delays. If this were a problem, one could imagine adopting a hybrid scheme that switched tactics if overload were detected.

The caching protocol is simple: it invalidates cached data on a change, and never updates it. It would be just as simple to update rather than to invalidate, but update-only protocols can be arbitrarily inefficient; a client that accessed a file might receive updates indefinitely, causing an unbounded number of unnecessary updates.

Despite the overheads of providing strict consistency, we rejected weaker models because we felt that programmers would find them harder to use. Similarly, mechanisms such as virtual synchrony that require clients to exchange sequence numbers in all messages were considered inappropriate in an environment with diverse pre-existing communication protocols.

In addition to caching data and meta-data, Chubby clients cache open handles. Thus, if a client opens a file it has opened previously, only the first `Open()` call necessarily leads to an RPC to the master. This caching is restricted in minor ways so that it never affects the semantics observed by the client: handles on ephemeral files cannot be held open if the application has closed them; and handles that permit locking can be reused, but cannot be used concurrently by multiple application handles. This last restriction exists because the client may use `Close()` or `Poison()` for their side-effect of cancelling outstanding `Acquire()` calls to the master.

Chubby's protocol permits clients to cache locks—that is, to hold locks longer than strictly necessary in the hope that they can be used again by the same client. An event informs a lock holder if another client has requested a conflicting lock, allowing the holder to release the lock just when it is needed elsewhere (see §2.5).

2.8 Sessions and KeepAlives

A Chubby session is a relationship between a Chubby cell and a Chubby client; it exists for some interval of time, and is maintained by periodic handshakes called KeepAlives. Unless a Chubby client informs the master otherwise, the client's handles, locks, and cached data all remain valid provided its session remains valid. (However, the protocol for session maintenance may require the client to acknowledge a cache invalidation in order to maintain its session; see below.)

A client requests a new session on first contacting the master of a Chubby cell. It ends the session explicitly either when it terminates, or if the session has been idle (with no open handles and no calls for a minute).

Each session has an associated lease—an interval of time extending into the future during which the master guarantees not to terminate the session unilaterally. The end of this interval is called the session lease timeout. The master is free to advance this timeout further into the future, but may not move it backwards in time.

The master advances the lease timeout in three circumstances: on creation of the session, when a master fail-over occurs (see below), and when it responds to a KeepAlive RPC from the client. On receiving a KeepAlive, the master typically blocks the RPC (does not allow it to return) until the client's previous lease interval is close to expiring. The master later allows the RPC to return to the client, and thus informs the client of the new lease timeout. The master may extend the timeout by any amount. The default extension is 12s, but an overloaded master may use higher values to reduce the number of KeepAlive calls it must process. The client initiates a new KeepAlive immediately after receiving the previous reply. Thus, the client ensures that there is almost always a KeepAlive call blocked at the master.

As well as extending the client's lease, the KeepAlive reply is used to transmit events and cache invalidations back to the client. The master allows a KeepAlive to return early when an event or invalidation is to be delivered. Piggybacking events on KeepAlive replies ensures that clients cannot maintain a session without acknowledging cache invalidations, and causes all Chubby RPCs to flow from client to master. This simplifies the client, and allows the protocol to operate through firewalls that allow initiation of connections in only one direction.

The client maintains a local lease timeout that is a conservative approximation of the master's lease timeout. It differs from the master's lease timeout because the client must make conservative assumptions both of the time its KeepAlive reply spent in flight, and the rate at which the master's clock is advancing; to maintain consistency, we require that the server's clock advance no faster than a known constant factor faster than the client's.

If a client's local lease timeout expires, it becomes unsure whether the master has terminated its session. The client empties and disables its cache, and we say that its session is in *jeopardy*. The client waits a further interval called the grace period, 45s by default. If the client and master manage to exchange a successful KeepAlive be-

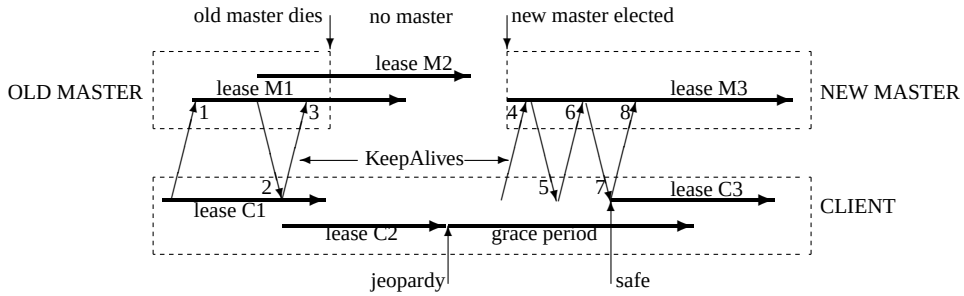


Figure 2: The role of the grace period in master fail-over

fore the end of the client's grace period, the client enables its cache once more. Otherwise, the client assumes that the session has expired. This is done so that Chubby API calls do not block indefinitely when a Chubby cell becomes inaccessible; calls return with an error if the grace period ends before communication is re-established.

The Chubby library can inform the application when the grace period begins via a *jeopardy* event. When the session is known to have survived the communications problem, a *safe* event tells the client to proceed; if the session times out instead, an *expired* event is sent. This information allows the application to quiesce itself when it is unsure of the status of its session, and to recover without restarting if the problem proves to be transient. This can be important in avoiding outages in services with large startup overhead.

If a client holds a handle H on a node and any operation on H fails because the associated session has expired, all subsequent operations on H (except `Close()` and `Poison()`) will fail in the same way. Clients can use this to guarantee that network and server outages cause only a suffix of a sequence of operations to be lost, rather than an arbitrary subsequence, thus allowing complex changes to be marked as committed with a final write.

2.9 Fail-overs

When a master fails or otherwise loses mastership, it discards its in-memory state about sessions, handles, and locks. The authoritative timer for session leases runs at the master, so until a new master is elected the session lease timer is stopped; this is legal because it is equivalent to extending the client's lease. If a master election occurs quickly, clients can contact the new master before their local (approximate) lease timers expire. If the election takes a long time, clients flush their caches and wait for the grace period while trying to find the new master. Thus the grace period allows sessions to be maintained across fail-overs that exceed the normal lease timeout.

Figure 2 shows the sequence of events in a lengthy master fail-over event in which the client must use its grace period to preserve its session. Time increases from left to right, but times are not to scale. Client session leases are shown as thick arrows both as viewed by both the old and new masters (M1-3, above) and the client (C1-3, below). Upward angled arrows indicate KeepAlive requests, and downward angled arrows their replies. The original master has session lease M1 for the client, while the client has a conservative approximation C1. The master commits to lease M2 before informing the client via KeepAlive reply 2; the client is able to extend its view of the lease C2. The master dies before replying to the next KeepAlive, and some time elapses before another master is elected. Eventually the client's approximation of its lease (C2) expires. The client then flushes its cache and starts a timer for the grace period.

During this period, the client cannot be sure whether its lease has expired at the master. It does not tear down its session, but it blocks all application calls on its API to prevent the application from observing inconsistent data. At the start of the grace period, the Chubby library sends a *jeopardy* event to the application to allow it to quiesce itself until it can be sure of the status of its session.

Eventually a new master election succeeds. The master initially uses a conservative approximation M3 of the session lease that its predecessor may have had for the client. The first KeepAlive request (4) from the client to the new master is rejected because it has the wrong master epoch number (described in detail below). The retried request (6) succeeds but typically does not extend the master lease further because M3 was conservative. However the reply (7) allows the client to extend its lease (C3) once more, and optionally inform the application that its session is no longer in jeopardy. Because the grace period was long enough to cover the interval between the end of lease C2 and the beginning of lease C3, the client saw nothing but a delay. Had the grace period been less than that interval, the client would have abandoned the session and reported the failure to the application.

Once a client has contacted the new master, the client library and master co-operate to provide the illusion to the application that no failure has occurred. To achieve this, the new master must reconstruct a conservative approximation of the in-memory state that the previous master had. It does this partly by reading data stored stably on disc (replicated via the normal database repli-

cation protocol), partly by obtaining state from clients, and partly by conservative assumptions. The database records each session, held lock, and ephemeral file.

A newly elected master proceeds:

1. It first picks a new client *epoch number*, which clients are required to present on every call. The master rejects calls from clients using older epoch numbers, and provides the new epoch number. This ensures that the new master will not respond to a very old packet that was sent to a previous master, even one running on the same machine.
2. The new master may respond to master-location requests, but does not at first process incoming session-related operations.
3. It builds in-memory data structures for sessions and locks that are recorded in the database. Session leases are extended to the maximum that the previous master may have been using.
4. The master now lets clients perform KeepAlives, but no other session-related operations.
5. It emits a fail-over event to each session; this causes clients to flush their caches (because they may have missed invalidations), and to warn applications that other events may have been lost.
6. The master waits until each session acknowledges the fail-over event or lets its session expire.
7. The master allows all operations to proceed.
8. If a client uses a handle created prior to the fail-over (determined from the value of a sequence number in the handle), the master recreates the in-memory representation of the handle and honours the call. If such a recreated handle is closed, the master records it in memory so that it cannot be recreated in this master epoch; this ensures that a delayed or duplicated network packet cannot accidentally recreate a closed handle. A faulty client can recreate a closed handle in a future epoch, but this is harmless given that the client is already faulty.
9. After some interval (a minute, say), the master deletes ephemeral files that have no open file handles. Clients should refresh handles on ephemeral files during this interval after a fail-over. This mechanism has the unfortunate effect that ephemeral files may not disappear promptly if the last client on such a file loses its session during a fail-over.

Readers will be unsurprised to learn that the fail-over code, which is exercised far less often than other parts of the system, has been a rich source of interesting bugs.

2.10 Database implementation

The first version of Chubby used the replicated version of Berkeley DB [20] as its database. Berkeley DB provides B-trees that map byte-string keys to arbitrary byte-string values. We installed a key comparison function that sorts first by the number of components in a path name; this allows nodes to by keyed by their path name, while keeping sibling nodes adjacent in the sort order. Because Chubby does not use path-based permissions, a single lookup in the database suffices for each file access.

Berkeley DB's uses a distributed consensus protocol to replicate its database logs over a set of servers. Once master leases were added, this matched the design of Chubby, which made implementation straightforward.

While Berkeley DB's B-tree code is widely-used and mature, the replication code was added recently, and has fewer users. Software maintainers must give priority to maintaining and improving their most popular product features. While Berkeley DB's maintainers solved the problems we had, we felt that use of the replication code exposed us to more risk than we wished to take. As a result, we have written a simple database using write ahead logging and snapshotting similar to the design of Birrell *et al.* [2]. As before, the database log is distributed among the replicas using a distributed consensus protocol. Chubby used few of the features of Berkeley DB, and so this rewrite allowed significant simplification of the system as a whole; for example, while we needed atomic operations, we did not need general transactions.

2.11 Backup

Every few hours, the master of each Chubby cell writes a snapshot of its database to a GFS file server [7] in a different building. The use of a separate building ensures both that the backup will survive building damage, and that the backups introduce no cyclic dependencies in the system; a GFS cell in the same building potentially might rely on the Chubby cell for electing its master.

Backups provide both disaster recovery and a means for initializing the database of a newly replaced replica without placing load on replicas that are in service.

2.12 Mirroring

Chubby allows a collection of files to be mirrored from one cell to another. Mirroring is fast because the files are small and the event mechanism (§2.5) informs the mirroring code immediately if a file is added, deleted, or modified. Provided there are no network problems, changes are reflected in dozens of mirrors world-wide in well under a second. If a mirror is unreachable, it remains unchanged until connectivity is restored. Updated files are then identified by comparing their checksums.

Mirroring is used most commonly to copy configuration files to various computing clusters distributed around the world. A special cell, named `global`, contains a subtree `/ls/global/master` that is mirrored to the

subtree `/ls/`*cell*`/slave` in every other Chubby cell. The global cell is special because its five replicas are located in widely-separated parts of the world, so it is almost always accessible from most of the organization.

Among the files mirrored from the global cell are Chubby's own access control lists, various files in which Chubby cells and other systems advertise their presence to our monitoring services, pointers to allow clients to locate large data sets such as Bigtable cells, and many configuration files for other systems.

3 Mechanisms for scaling

Chubby's clients are individual processes, so Chubby must handle more clients than one might expect; we have seen 90,000 clients communicating directly with a Chubby master—far more than the number of machines involved. Because there is just one master per cell, and its machine is identical to those of the clients, the clients can overwhelm the master by a huge margin. Thus, the most effective scaling techniques reduce communication with the master by a significant factor. Assuming the master has no serious performance bug, minor improvements in request processing at the master have little effect. We use several approaches:

- We can create an arbitrary number of Chubby cells; clients almost always use a nearby cell (found with DNS) to avoid reliance on remote machines. Our typical deployment uses one Chubby cell for a data centre of several thousand machines.
- The master may increase lease times from the default 12s up to around 60s when it is under heavy load, so it need process fewer KeepAlive RPCs. (KeepAlives are *by far* the dominant type of request (see 4.1), and failure to process them in time is the typical failure mode of an overloaded server; clients are largely insensitive to latency variation in other calls.)
- Chubby clients cache file data, meta-data, the absence of files, and open handles to reduce the number of calls they make on the server.
- We use protocol-conversion servers that translate the Chubby protocol into less-complex protocols such as DNS and others. We discuss some of these below.

Here we describe two familiar mechanisms, proxies and partitioning, that we expect will allow Chubby to scale further. We do not yet use them in production, but they are designed, and may be used soon. We have no present need to consider scaling beyond a factor of five: First, there are limits on the number of machines one would wish to put in a data centre or make reliant on a single instance of a service. Second, because we use similar machines for Chubby clients and servers, hardware improvements that increase the number of clients per machine also increase the capacity of each server.

3.1 Proxies

Chubby's protocol can be proxied (using the same protocol on both sides) by trusted processes that pass requests from other clients to a Chubby cell. A proxy can reduce server load by handling both KeepAlive and read requests; it cannot reduce write traffic, which passes through the proxy's cache. But even with aggressive client caching, write traffic constitutes much less than one percent of Chubby's normal workload (see §4.1), so proxies allow a significant increase in the number of clients. If a proxy handles N_{proxy} clients, KeepAlive traffic is reduced by a factor of N_{proxy}, which might be 10 thousand or more. A proxy cache can reduce read traffic by at most the mean amount of read-sharing—a factor of around 10 (§4.1). But because reads constitute under 10% of Chubby's load at present, the saving in KeepAlive traffic is by far the more important effect.

Proxies add an additional RPC to writes and first-time reads. One might expect proxies to make the cell temporarily unavailable at least twice as often as before, because each proxied client depends on two machines that may fail: its proxy and the Chubby master.

Alert readers will notice that the fail-over strategy described in Section 2.9, is not ideal for proxies. We discuss this problem in Section 4.4.

3.2 Partitioning

As mentioned in Section 2.3, Chubby's interface was chosen so that the name space of a cell could be partitioned between servers. Although we have not yet needed it, the code can partition the name space by directory. If enabled, a Chubby cell would be composed of N *partitions*, each of which has a set of replicas and a master. Every node D/C in directory D would be stored on the partition $P(D/C) = \texttt{hash}(D) \texttt{ mod } N$. Note that the meta-data for D may be stored on a different partition $P(D) = \texttt{hash}(D') \texttt{ mod } N$, where D' is the parent of D.

Partitioning is intended to enable large Chubby cells with little communication between the partitions. Although Chubby lacks hard links, directory modified-times, and cross-directory rename operations, a few operations still require cross-partition communication:

- ACLs are themselves files, so one partition may use another for permissions checks. However, ACL files are readily cached; only `Open()` and `Delete()` calls require ACL checks; and most clients read publicly accessible files that require no ACL.
- When a directory is deleted, a cross-partition call may be needed to ensure that the directory is empty.

Because each partition handles most calls independently of the others, we expect this communication to have only a modest impact on performance or availability.

Unless the number of partitions N is large, one would expect that each client would contact the majority of the partitions. Thus, partitioning reduces read and write traffic on any given partition by a factor of N but does not necessarily reduce KeepAlive traffic. Should it be necessary for Chubby to handle more clients, our strategy involves a combination of proxies and partitioning.

4 Use, surprises and design errors

4.1 Use and behaviour

The following table gives statistics taken as a snapshot of a Chubby cell; the RPC rate was a seen over a ten-minute period. The numbers are typical of cells in Google.

time since last fail-over	18 days
fail-over duration	14s
active clients (direct)	22k
additional proxied clients	32k
files open	12k
naming-related	60%
client-is-caching-file entries	230k
distinct files cached	24k
names negatively cached	32k
exclusive locks	1k
shared locks	0
stored directories	8k
ephemeral	0.1%
stored files	22k
0-1k bytes	90%
1k-10k bytes	10%
$>$ 10k bytes	0.2%
naming-related	46%
mirrored ACLs & config info	27%
GFS and Bigtable meta-data	11%
ephemeral	3%
RPC rate	1-2k/s
KeepAlive	93%
GetStat	2%
Open	1%
CreateSession	1%
GetContentsAndStat	0.4%
SetContents	680ppm
Acquire	31ppm

Several things can be seen:

- Many files are used for naming; see §4.3.
- Configuration, access control, and meta-data files (analogous to file system super-blocks) are common.
- Negative caching is significant.
- 230k/24k≈10 clients use each cached file, on average.
- Few clients hold locks, and shared locks are rare; this is consistent with locking being used for primary election and partitioning data among replicas.
- RPC traffic is dominated by session KeepAlives; there are a few reads (which are cache misses); there are very few writes or lock acquisitions.

Now we briefly describe the typical causes of outages in our cells. If we assume (optimistically) that a cell is "up" if it has a master that is willing to serve, on a sample of our cells we recorded 61 outages over a period of a few weeks, amounting to 700 cell-days of data in total. We excluded outages due to maintenance that shut down the data centre. All other causes are included: network congestion, maintenance, overload, and errors due to operators, software, and hardware. Most outages were 15s or less, and 52 were under 30s; most of our applications are not affected significantly by Chubby outages under 30s. The remaining nine outages were caused by network maintenance (4), suspected network connectivity problems (2), software errors (2), and overload (1).

In a few dozen cell-years of operation, we have lost data on six occasions, due to database software errors (4) and operator error (2); none involved hardware error. Ironically, the operational errors involved upgrades to avoid the software errors. We have twice corrected corruptions caused by software in non-master replicas.

Chubby's data fits in RAM, so most operations are cheap. Mean request latency at our production servers is consistently a small fraction of a millisecond regardless of cell load until the cell approaches overload, when latency increases dramatically and sessions are dropped. Overload typically occurs when many sessions ($> 90,000$) are active, but can result from exceptional conditions: when clients made millions of read requests simultaneously (described in Section 4.3), and when a mistake in the client library disabled caching for some reads, resulting in tens of thousands of requests per second. Because most RPCs are KeepAlives, the server can maintain a low mean request latency with many active clients by increasing the session lease period (see §3). Group commit reduces the effective work done per request when bursts of writes arrive, but this is rare.

RPC read latencies measured at the client are limited by the RPC system and network; they are under 1ms for a local cell, but 250ms between antipodes. Writes (which include lock operations) are delayed a further 5-10ms by the database log update, but by up to tens of seconds if a recently-failed client cached the file. Even this variability in write latency has little effect on the mean request latency at the server because writes are so infrequent.

Clients are fairly insensitive to latency variation provided sessions are not dropped. At one point, we added artificial delays in `Open()` to curb abusive clients (see §4.5); developers noticed only when delays exceeded ten seconds *and* were applied repeatedly. We have found that the key to scaling Chubby is not server performance; reducing communication to the server can have far greater

impact. No significant effort has been applied to tuning read/write server code paths; we checked that no egregious bugs were present, then focused on the scaling mechanisms that could be more effective. On the other hand, developers do notice if a performance bug affects the local Chubby cache, which a client may read thousands of times per second.

4.2 Java clients

Google's infrastructure is mostly in C++, but a growing number of systems are being written in Java [8]. This trend presented an unanticipated problem for Chubby, which has a complex client protocol and a non-trivial client-side library.

Java encourages portability of entire applications at the expense of incremental adoption by making it somewhat irksome to link against other languages. The usual Java mechanism for accessing non-native libraries is JNI [15], but it is regarded as slow and cumbersome. Our Java programmers so dislike JNI that to avoid its use they prefer to translate large libraries into Java, and commit to supporting them.

Chubby's C++ client library is 7000 lines (comparable with the server), and the client protocol is delicate. To maintain the library in Java would require care and expense, while an implementation without caching would burden the Chubby servers. Thus our Java users run copies of a protocol-conversion server that exports a simple RPC protocol that corresponds closely to Chubby's client API. Even with hindsight, it is not obvious how we might have avoided the cost of writing, running and maintaining this additional server.

4.3 Use as a name service

Even though Chubby was designed as a lock service, we found that its most popular use was as a name server.

Caching within the normal Internet naming system, the DNS, is based on time. DNS entries have a *time-to-live* (TTL), and DNS data are discarded when they have not been refreshed within that period. Usually it is straightforward to pick a suitable TTL value, but if prompt replacement of failed services is desired, the TTL can become small enough to overload the DNS servers.

For example, it is common for our developers to run jobs involving thousands of processes, and for each process to communicate with every other, leading to a quadratic number of DNS lookups. We might wish to use a TTL of 60s; this would allow misbehaving clients to be replaced without excessive delay and is not considered an unreasonably short replacement time in our environment. In that case, to maintain the DNS caches of a single job as small as 3 thousand clients would require 150 thousand lookups per second. (For comparison, a 2-CPU 2.6GHz Xeon DNS server might handle 50 thousand requests per second.) Larger jobs create worse problems, and several jobs many be running at once. The variability in our DNS load had been a serious problem for Google before Chubby was introduced.

In contrast, Chubby's caching uses explicit invalidations so a constant rate of session KeepAlive requests can maintain an arbitrary number of cache entries indefinitely at a client, in the absence of changes. A 2-CPU 2.6GHz Xeon Chubby master has been seen to handle 90 thousand clients communicating directly with it (no proxies); the clients included large jobs with communication patterns of the kind described above. The ability to provide swift name updates without polling each name individually is so appealing that Chubby now provides name service for most of the company's systems.

Although Chubby's caching allows a single cell to sustain a large number of clients, load spikes can still be a problem. When we first deployed the Chubby-based name service, starting a 3 thousand process job (thus generating 9 million requests) could bring the Chubby master to its knees. To resolve this problem, we chose to group name entries into batches so that a single lookup would return and cache the name mappings for a large number (typically 100) of related processes within a job.

The caching semantics provided by Chubby are more precise than those needed by a name service; name resolution requires only timely notification rather than full consistency. As a result, there was an opportunity for reducing the load on Chubby by introducing a simple protocol-conversion server designed specifically for name lookups. Had we foreseen the use of Chubby as a name service, we might have chosen to implement full proxies sooner than we did in order to avoid the need for this simple, but nevertheless additional server.

One further protocol-conversion server exists: the Chubby DNS server. This makes the naming data stored within Chubby available to DNS clients. This server is important both for easing the transition from DNS names to Chubby names, and to accommodate existing applications that cannot be converted easily, such as browsers.

4.4 Problems with fail-over

The original design for master fail-over (§2.9) requires the master to write new sessions to the database as they are created. In the Berkeley DB version of the lock server, the overhead of creating sessions became a problem when many processes were started at once. To avoid overload, the server was modified to store a session in the database not when it was first created, but instead when it attempted its first modification, lock acquisition, or open

of an ephemeral file. In addition, active sessions were recorded in the database with some probability on each KeepAlive. Thus, the writes for read-only sessions were spread out in time.

Though it was necessary to avoid overload, this optimization has the undesirable effect that young read-only sessions may not be recorded in the database, and so may be discarded if a fail-over occurs. Although such sessions hold no locks, this is unsafe; if all the recorded sessions were to check in with the new master before the leases of discarded sessions expired, the discarded sessions could then read stale data for a while. This is rare in practice; in a large system it is almost certain that some session will fail to check in, and thus force the new master to await the maximum lease time anyway. Nevertheless, we have modified the fail-over design both to avoid this effect, and to avoid a complication that the current scheme introduces to proxies.

Under the new design, we avoid recording sessions in the database at all, and instead recreate them in the same way that the master currently recreates handles (§2.9,¶8). A new master must now wait a full worst-case lease timeout before allowing operations to proceed, since it cannot know whether all sessions have checked in (§2.9,¶6). Again, this has little effect in practice because it is likely that not all sessions will check in.

Once sessions can be recreated without on-disc state, proxy servers can manage sessions that the master is not aware of. An extra operation available only to proxies allows them to change the session that locks are associated with. This permits one proxy to take over a client from another when a proxy fails. The only further change needed at the master is a guarantee not to relinquish locks or ephemeral file handles associated with proxy sessions until a new proxy has had a chance to claim them.

4.5 Abusive clients

Google's project teams are free to set up their own Chubby cells, but doing so adds to their maintenance burden, and consumes additional hardware resources. Many services therefore use shared Chubby cells, which makes it important to isolate clients from the misbehaviour of others. Chubby is intended to operate within a single company, and so malicious denial-of-service attacks against it are rare. However, mistakes, misunderstandings, and the differing expectations of our developers lead to effects that are similar to attacks.

Some of our remedies are heavy-handed. For example, we review the ways project teams plan to use Chubby, and deny access to the shared Chubby name space until review is satisfactory. A problem with this approach is that developers are often unable to predict how their services will be used in the future, and how use will grow. Readers will note the irony of our own failure to predict how Chubby itself would be used.

The most important aspect of our review is to determine whether use of any of Chubby's resources (RPC rate, disc space, number of files) grows linearly (or worse) with number of users or amount of data processed by the project. Any linear growth must be mitigated by a compensating parameter that can be adjusted to reduce the load on Chubby to reasonable bounds. Nevertheless our early reviews were not thorough enough.

A related problem is the lack of performance advice in most software documentation. A module written by one team may be reused a year later by another team with disastrous results. It is sometimes hard to explain to interface designers that they must change their interfaces not because they are bad, but because other developers may be less aware of the cost of an RPC.

Below we list some problem cases we encountered.

Lack of aggressive caching Originally, we did not appreciate the critical need to cache the absence of files, nor to reuse open file handles. Despite attempts at education, our developers regularly write loops that retry indefinitely when a file is not present, or poll a file by opening it and closing it repeatedly when one might expect they would open the file just once.

At first we countered these retry-loops by introducing exponentially-increasing delays when an application made many attempts to `Open()` the same file over a short period. In some cases this exposed bugs that developers acknowledged, but often it required us to spend yet more time on education. In the end it was easier to make repeated `Open()` calls cheap.

Lack of quotas Chubby was never intended to be used as a storage system for large amounts of data, and so it has no storage quotas. In hindsight, this was naïve.

One of Google's projects wrote a module to keep track of data uploads, storing some meta-data in Chubby. Such uploads occurred rarely and were limited to a small set of people, so the space was bounded. However, two other services started using the same module as a means for tracking uploads from a wider population of users. Inevitably, these services grew until the use of Chubby was extreme: a single 1.5MByte file was being rewritten in its entirety on each user action, and the overall space used by the service exceeded the space needs of all other Chubby clients combined.

We introduced a limit on file size (256kBytes), and encouraged the services to migrate to more appropriate storage systems. But it is difficult to make significant changes to production systems maintained by busy people—it took approximately a year for the data to be migrated elsewhere.

Publish/subscribe There have been several attempts to use Chubby's event mechanism as a publish/subscribe

system in the style of Zephyr [6]. Chubby's heavyweight guarantees and its use of invalidation rather than update in maintaining cache consistency make it a slow and inefficient for all but the most trivial publish/subscribe examples. Fortunately, all such uses have been caught before the cost of redesigning the application was too large.

4.6 Lessons learned

Here we list lessons, and miscellaneous design changes we might make if we have the opportunity:

Developers rarely consider availability We find that our developers rarely think about failure probabilities, and are inclined to treat a service like Chubby as though it were always available. For example, our developers once built a system employing hundred of machines that initiated recovery procedures taking tens of minutes when Chubby elected a new master. This magnified the consequences of a single failure by a factor of a hundred both in time *and* the number of machines affected. We would prefer developers to plan for short Chubby outages, so that such an event has little or no affect on their applications. This is one of the arguments for coarse-grained locking, discussed in Section 2.1.

Developers also fail to appreciate the difference between a service being up, and that service being available to their applications. For example, the global Chubby cell (see §2.12), is almost always up because it is rare for more than two geographically distant data centres to be down simultaneously. However, its *observed availability for a given client* is usually lower than the observed availability of the client's local Chubby cell. First, the local cell is less likely to be partitioned from the client, and second, although the local cell may be down often due to maintenance, the same maintenance affects the client directly, so Chubby's unavailability is not observed by the client.

Our API choices can also affect the way developers chose to handle Chubby outages. For example, Chubby provides an event that allows clients to detect when a master fail-over has taken place. The intent was for clients to check for possible changes, as other events may have been lost. Unfortunately, many developers chose to crash their applications on receiving this event, thus decreasing the availability of their systems substantially. We might have done better to send redundant "file change" events instead, or even to ensure that no events were lost during a fail-over.

At present we use three mechanisms to prevent developers from being over-optimistic about Chubby availability, especially that of the global cell. First, as previously mentioned (§4.5), we review how project teams plan to use Chubby, and advise them against techniques that would tie their availability too closely to Chubby's. Second, we now supply libraries that perform some high-level tasks so that developers are automatically isolated from Chubby outages. Third, we use the post-mortem of each Chubby outage as a means not only of eliminating bugs in Chubby and our operational procedures, but of reducing the sensitivity of applications to Chubby's availability—both can lead to better availability of our systems overall.

Fine-grained locking could be ignored At the end of Section 2.1 we sketched a design for a server that clients could run to provide fine-grained locking. It is perhaps a surprise that so far we have not needed to write such a server; our developers typically find that to optimize their applications, they must remove unnecessary communication, and that often means finding a way to use coarse-grained locking.

Poor API choices have unexpected affects For the most part, our API has evolved well, but one mistake stands out. Our means for cancelling long-running calls are the `Close()` and `Poison()` RPCs, which also discard the server state for the handle. This prevents handles that can acquire locks from being shared, for example, by multiple threads. We may add a `Cancel()` RPC to allow more sharing of open handles.

RPC use affects transport protocols KeepAlives are used both for refreshing the client's session lease, and for passing events and cache invalidations from the master to the client. This design has the automatic and desirable consequence that a client cannot refresh its session lease without acknowledging cache invalidations.

This would seem ideal, except that it introduced a tension in our choice of protocol. TCP's back off policies pay no attention to higher-level timeouts such as Chubby leases, so TCP-based KeepAlives led to many lost sessions at times of high network congestion. We were forced to send KeepAlive RPCs via UDP rather than TCP; UDP has no congestion avoidance mechanisms, so we would prefer to use UDP only when high-level timebounds must be met.

We may augment the protocol with an additional TCP-based `GetEvent()` RPC which would be used to communicate events and invalidations in the normal case, used in the same way KeepAlives. The KeepAlive reply would still contain a list of unacknowledged events so that events must eventually be acknowledged.

5 Comparison with related work

Chubby is based on well-established ideas. Chubby's cache design is derived from work on distributed file systems [10]. Its sessions and cache tokens are similar in behaviour to those in Echo [17]; sessions reduce the overhead of leases [9] in the V system. The idea of exposing a general-purpose lock service is found in VMS [23],

though that system initially used a special-purpose high-speed interconnect that permitted low-latency interactions. Like its caching model, Chubby's API is based on a file-system model, including the idea that a file-system-like name space is convenient for more than just files [18, 21, 22].

Chubby differs from a distributed file system such as Echo or AFS [10] in its performance and storage aspirations: Clients do not read, write, or store large amounts of data, and they do not expect high throughput or even low-latency unless the data is cached. They do expect consistency, availability, and reliability, but these attributes are easier to achieve when performance is less important. Because Chubby's database is small, we are able to store many copies of it on-line (typically five replicas and a few backups). We take full backups multiple times per day, and via checksums of the database state, we compare replicas with one another every few hours. The weakening of the normal file system performance and storage requirements allows us to serve tens of thousands of clients from a single Chubby master. By providing a central point where many clients can share information and co-ordinate activities, we have solved a class of problems faced by our system developers.

The large number of file systems and lock servers described in the literature prevents an exhaustive comparison, so we provide details on one: we chose to compare with Boxwood's lock server [16] because it was designed recently, it too is designed to run in a loosely-coupled environment, and yet its design differs in various ways from Chubby, some interesting and some incidental.

Chubby implements locks, a reliable small-file storage system, and a session/lease mechanism in a single service. In contrast, Boxwood separates these into three: a *lock service*, a *Paxos service* (a reliable repository for state), and a *failure detection service* respectively. The Boxwood system itself uses these three components together, but another system could use these building blocks independently. We suspect that this difference in design stems from a difference in target audience. Chubby was intended for a diverse audience and application mix; its users range from experts who create new distributed systems, to novices who write administration scripts. For our environment, a large-scale shared service with a familiar API seemed attractive. In contrast, Boxwood provides a toolkit that (to our eyes, at least) is appropriate for a smaller number of more sophisticated developers working on projects that may share code but need not be used together.

In many cases, Chubby provides a higher-level interface than Boxwood. For example, Chubby combines the lock and file names spaces, while Boxwood's lock names are simple byte sequences. Chubby clients cache file state by default; a client of Boxwood's Paxos service could implement caching via the lock service, but would probably use the caching provided by Boxwood itself.

The two systems have markedly different default parameters, chosen for different expectations: Each Boxwood failure detector is contacted by each client every 200ms with a timeout of 1s; Chubby's default lease time is 12s and KeepAlives are exchanged every 7s. Boxwood's subcomponents use two or three replicas to achieve availability, while we typically use five replicas per cell. However, these choices alone do not suggest a deep design difference, but rather an indication of how parameters in such systems must be adjusted to accommodate more client machines, or the uncertainties of racks shared with other projects.

A more interesting difference is the introduction of Chubby's grace period, which Boxwood lacks. (Recall that the grace period allows clients to ride out long Chubby master outages without losing sessions or locks. Boxwood's "grace period" is the equivalent of Chubby's "session lease", a different concept.) Again, this difference is the result of differing expectations about scale and failure probability in the two systems. Although master fail-overs are rare, a lost Chubby lock is expensive for clients.

Finally, locks in the two systems are intended for different purposes. Chubby locks are heavier-weight, and need sequencers to allow externals resources to be protected safely, while Boxwood locks are lighter-weight, and intended primarily for use within Boxwood.

6 Summary

Chubby is a distributed lock service intended for coarse-grained synchronization of activities within Google's distributed systems; it has found wider use as a name service and repository for configuration information.

Its design is based on well-known ideas that have meshed well: distributed consensus among a few replicas for fault tolerance, consistent client-side caching to reduce server load while retaining simple semantics, timely notification of updates, and a familiar file system interface. We use caching, protocol-conversion servers, and simple load adaptation to allow it scale to tens of thousands of client processes per Chubby instance. We expect to scale it further via proxies and partitioning.

Chubby has become Google's primary internal name service; it is a common rendezvous mechanism for systems such as MapReduce [4]; the storage systems GFS and Bigtable use Chubby to elect a primary from redundant replicas; and it is a standard repository for files that require high availability, such as access control lists.

7 Acknowledgments

Many contributed to the Chubby system: Sharon Perl wrote the replication layer on Berkeley DB; Tushar Chandra and Robert Griesemer wrote the replicated database that replaced Berkeley DB; Ramsey Haddad connected the API to Google's file system interface; Dave Presotto, Sean Owen, Doug Zongker and Praveen Tamara wrote the Chubby DNS, Java, and naming protocol-converters, and the full Chubby proxy respectively; Vadim Furman added the caching of open handles and file-absence; Rob Pike, Sean Quinlan and Sanjay Ghemawat gave valuable design advice; and many Google developers uncovered early weaknesses.

References

[1] BIRMAN, K. P., AND JOSEPH, T. A. Exploiting virtual synchrony in distributed systems. In *11th SOSP* (1987), pp. 123–138.

[2] BIRRELL, A., JONES, M. B., AND WOBBER, E. A simple and efficient implementation for small databases. In *11th SOSP* (1987), pp. 149–154.

[3] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. Bigtable: A distributed structured data storage system. In *7th OSDI* (2006).

[4] DEAN, J., AND GHEMAWAT, S. MapReduce: Simplified data processing on large clusters. In *6th OSDI* (2004), pp. 137–150.

[5] FISCHER, M. J., LYNCH, N. A., AND PATERSON, M. S. Impossibility of distributed consensus with one faulty process. *J. ACM 32*, 2 (April 1985), 374–382.

[6] FRENCH, R. S., AND KOHL, J. T. *The Zephyr Programmer's Manual*. MIT Project Athena, Apr. 1989.

[7] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The Google file system. In *19th SOSP* (Dec. 2003), pp. 29–43.

[8] GOSLING, J., JOY, B., STEELE, G., AND BRACHA, G. *Java Language Spec. (2nd Ed.)*. Addison-Wesley, 2000.

[9] GRAY, C. G., AND CHERITON, D. R. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *12th SOSP* (1989), pp. 202–210.

[10] HOWARD, J., KAZAR, M., MENEES, S., NICHOLS, D., SATYANARAYANAN, M., SIDEBOTHAM, R., AND WEST, M. Scale and performance in a distributed file system. *ACM TOCS 6*, 1 (Feb. 1988), 51–81.

[11] JEFFERSON, D. Virtual time. *ACM TOPLAS*, 3 (1985), 404–425.

[12] LAMPORT, L. The part-time parliament. *ACM TOCS 16*, 2 (1998), 133–169.

[13] LAMPORT, L. Paxos made simple. *ACM SIGACT News 32*, 4 (2001), 18–25.

[14] LAMPSON, B. W. How to build a highly available system using consensus. In *Distributed Algorithms*, vol. 1151 of *LNCS*. Springer–Verlag, 1996, pp. 1–17.

[15] LIANG, S. *Java Native Interface: Programmer's Guide and Reference*. Addison-Wesley, 1999.

[16] MACCORMICK, J., MURPHY, N., NAJORK, M., THEKKATH, C. A., AND ZHOU, L. Boxwood: Abstractions as the foundation for storage infrastructure. In *6th OSDI* (2004), pp. 105–120.

[17] MANN, T., BIRRELL, A., HISGEN, A., JERIAN, C., AND SWART, G. A coherent distributed file cache with directory write-behind. *TOCS 12*, 2 (1994), 123–164.

[18] MCJONES, P., AND SWART, G. Evolving the UNIX system interface to support multithreaded programs. Tech. Rep. 21, DEC SRC, 1987.

[19] OKI, B., AND LISKOV, B. Viewstamped replication: A general primary copy method to support highly-available distributed systems. In *ACM PODC* (1988).

[20] OLSON, M. A., BOSTIC, K., AND SELTZER, M. Berkeley DB. In *USENIX* (June 1999), pp. 183–192.

[21] PIKE, R., PRESOTTO, D. L., DORWARD, S., FLANDRENA, B., THOMPSON, K., TRICKEY, H., AND WINTERBOTTOM, P. Plan 9 from Bell Labs. *Computing Systems 8*, 2 (1995), 221–254.

[22] RITCHIE, D. M., AND THOMPSON, K. The UNIX time-sharing system. *CACM 17*, 7 (1974), 365–375.

[23] SNAMAN, JR., W. E., AND THIEL, D. W. The VAX/VMS distributed lock manager. *Digital Technical Journal 1*, 5 (Sept. 1987), 29–44.

[24] YIN, J., MARTIN, J.-P., VENKATARAMANI, A., ALVISI, L., AND DAHLIN, M. Separating agreement from execution for byzantine fault tolerant services. In *19th SOSP* (2003), pp. 253–267.

Chapter 12

The Google File System

Authors:
Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung

Publication year: 2003

The Google File System

Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung

Google*

ABSTRACT

We have designed and implemented the Google File System, a scalable distributed file system for large distributed data-intensive applications. It provides fault tolerance while running on inexpensive commodity hardware, and it delivers high aggregate performance to a large number of clients.

While sharing many of the same goals as previous distributed file systems, our design has been driven by observations of our application workloads and technological environment, both current and anticipated, that reflect a marked departure from some earlier file system assumptions. This has led us to reexamine traditional choices and explore radically different design points.

The file system has successfully met our storage needs. It is widely deployed within Google as the storage platform for the generation and processing of data used by our service as well as research and development efforts that require large data sets. The largest cluster to date provides hundreds of terabytes of storage across thousands of disks on over a thousand machines, and it is concurrently accessed by hundreds of clients.

In this paper, we present file system interface extensions designed to support distributed applications, discuss many aspects of our design, and report measurements from both micro-benchmarks and real world use.

Categories and Subject Descriptors

D [**4**]: 3—*Distributed file systems*

General Terms

Design, reliability, performance, measurement

Keywords

Fault tolerance, scalability, data storage, clustered storage

*The authors can be reached at the following addresses: {*sanjay,hgobioff,shuntak*}*@google.com*.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SOSP'03, October 19–22, 2003, Bolton Landing, New York, USA.
Copyright 2003 ACM 1-58113-757-5/03/0010 ...$5.00.

1. INTRODUCTION

We have designed and implemented the Google File System (GFS) to meet the rapidly growing demands of Google's data processing needs. GFS shares many of the same goals as previous distributed file systems such as performance, scalability, reliability, and availability. However, its design has been driven by key observations of our application workloads and technological environment, both current and anticipated, that reflect a marked departure from some earlier file system design assumptions. We have reexamined traditional choices and explored radically different points in the design space.

First, component failures are the norm rather than the exception. The file system consists of hundreds or even thousands of storage machines built from inexpensive commodity parts and is accessed by a comparable number of client machines. The quantity and quality of the components virtually guarantee that some are not functional at any given time and some will not recover from their current failures. We have seen problems caused by application bugs, operating system bugs, human errors, and the failures of disks, memory, connectors, networking, and power supplies. Therefore, constant monitoring, error detection, fault tolerance, and automatic recovery must be integral to the system.

Second, files are huge by traditional standards. Multi-GB files are common. Each file typically contains many application objects such as web documents. When we are regularly working with fast growing data sets of many TBs comprising billions of objects, it is unwieldy to manage billions of approximately KB-sized files even when the file system could support it. As a result, design assumptions and parameters such as I/O operation and block sizes have to be revisited.

Third, most files are mutated by appending new data rather than overwriting existing data. Random writes within a file are practically non-existent. Once written, the files are only read, and often only sequentially. A variety of data share these characteristics. Some may constitute large repositories that data analysis programs scan through. Some may be data streams continuously generated by running applications. Some may be archival data. Some may be intermediate results produced on one machine and processed on another, whether simultaneously or later in time. Given this access pattern on huge files, appending becomes the focus of performance optimization and atomicity guarantees, while caching data blocks in the client loses its appeal.

Fourth, co-designing the applications and the file system API benefits the overall system by increasing our flexibility.

For example, we have relaxed GFS's consistency model to vastly simplify the file system without imposing an onerous burden on the applications. We have also introduced an atomic append operation so that multiple clients can append concurrently to a file without extra synchronization between them. These will be discussed in more details later in the paper.

Multiple GFS clusters are currently deployed for different purposes. The largest ones have over 1000 storage nodes, over 300 TB of disk storage, and are heavily accessed by hundreds of clients on distinct machines on a continuous basis.

2. DESIGN OVERVIEW

2.1 Assumptions

In designing a file system for our needs, we have been guided by assumptions that offer both challenges and opportunities. We alluded to some key observations earlier and now lay out our assumptions in more details.

- The system is built from many inexpensive commodity components that often fail. It must constantly monitor itself and detect, tolerate, and recover promptly from component failures on a routine basis.
- The system stores a modest number of large files. We expect a few million files, each typically 100 MB or larger in size. Multi-GB files are the common case and should be managed efficiently. Small files must be supported, but we need not optimize for them.
- The workloads primarily consist of two kinds of reads: large streaming reads and small random reads. In large streaming reads, individual operations typically read hundreds of KBs, more commonly 1 MB or more. Successive operations from the same client often read through a contiguous region of a file. A small random read typically reads a few KBs at some arbitrary offset. Performance-conscious applications often batch and sort their small reads to advance steadily through the file rather than go back and forth.
- The workloads also have many large, sequential writes that append data to files. Typical operation sizes are similar to those for reads. Once written, files are seldom modified again. Small writes at arbitrary positions in a file are supported but do not have to be efficient.
- The system must efficiently implement well-defined semantics for multiple clients that concurrently append to the same file. Our files are often used as producer-consumer queues or for many-way merging. Hundreds of producers, running one per machine, will concurrently append to a file. Atomicity with minimal synchronization overhead is essential. The file may be read later, or a consumer may be reading through the file simultaneously.
- High sustained bandwidth is more important than low latency. Most of our target applications place a premium on processing data in bulk at a high rate, while few have stringent response time requirements for an individual read or write.

2.2 Interface

GFS provides a familiar file system interface, though it does not implement a standard API such as POSIX. Files are organized hierarchically in directories and identified by pathnames. We support the usual operations to *create*, *delete*, *open*, *close*, *read*, and *write* files.

Moreover, GFS has *snapshot* and *record append* operations. Snapshot creates a copy of a file or a directory tree at low cost. Record append allows multiple clients to append data to the same file concurrently while guaranteeing the atomicity of each individual client's append. It is useful for implementing multi-way merge results and producer-consumer queues that many clients can simultaneously append to without additional locking. We have found these types of files to be invaluable in building large distributed applications. Snapshot and record append are discussed further in Sections 3.4 and 3.3 respectively.

2.3 Architecture

A GFS cluster consists of a single *master* and multiple *chunkservers* and is accessed by multiple *clients*, as shown in Figure 1. Each of these is typically a commodity Linux machine running a user-level server process. It is easy to run both a chunkserver and a client on the same machine, as long as machine resources permit and the lower reliability caused by running possibly flaky application code is acceptable.

Files are divided into fixed-size *chunks*. Each chunk is identified by an immutable and globally unique 64 bit *chunk handle* assigned by the master at the time of chunk creation. Chunkservers store chunks on local disks as Linux files and read or write chunk data specified by a chunk handle and byte range. For reliability, each chunk is replicated on multiple chunkservers. By default, we store three replicas, though users can designate different replication levels for different regions of the file namespace.

The master maintains all file system metadata. This includes the namespace, access control information, the mapping from files to chunks, and the current locations of chunks. It also controls system-wide activities such as chunk lease management, garbage collection of orphaned chunks, and chunk migration between chunkservers. The master periodically communicates with each chunkserver in *HeartBeat* messages to give it instructions and collect its state.

GFS client code linked into each application implements the file system API and communicates with the master and chunkservers to read or write data on behalf of the application. Clients interact with the master for metadata operations, but all data-bearing communication goes directly to the chunkservers. We do not provide the POSIX API and therefore need not hook into the Linux vnode layer.

Neither the client nor the chunkserver caches file data. Client caches offer little benefit because most applications stream through huge files or have working sets too large to be cached. Not having them simplifies the client and the overall system by eliminating cache coherence issues. (Clients do cache metadata, however.) Chunkservers need not cache file data because chunks are stored as local files and so Linux's buffer cache already keeps frequently accessed data in memory.

2.4 Single Master

Having a single master vastly simplifies our design and enables the master to make sophisticated chunk placement

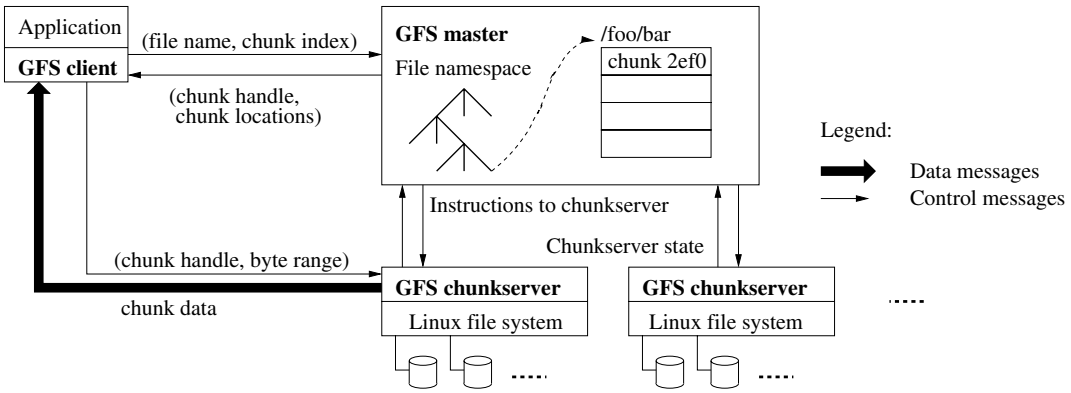


Figure 1: GFS Architecture

and replication decisions using global knowledge. However, we must minimize its involvement in reads and writes so that it does not become a bottleneck. Clients never read and write file data through the master. Instead, a client asks the master which chunkservers it should contact. It caches this information for a limited time and interacts with the chunkservers directly for many subsequent operations.

Let us explain the interactions for a simple read with reference to Figure 1. First, using the fixed chunk size, the client translates the file name and byte offset specified by the application into a chunk index within the file. Then, it sends the master a request containing the file name and chunk index. The master replies with the corresponding chunk handle and locations of the replicas. The client caches this information using the file name and chunk index as the key.

The client then sends a request to one of the replicas, most likely the closest one. The request specifies the chunk handle and a byte range within that chunk. Further reads of the same chunk require no more client-master interaction until the cached information expires or the file is reopened. In fact, the client typically asks for multiple chunks in the same request and the master can also include the information for chunks immediately following those requested. This extra information sidesteps several future client-master interactions at practically no extra cost.

2.5 Chunk Size

Chunk size is one of the key design parameters. We have chosen 64 MB, which is much larger than typical file system block sizes. Each chunk replica is stored as a plain Linux file on a chunkserver and is extended only as needed. Lazy space allocation avoids wasting space due to internal fragmentation, perhaps the greatest objection against such a large chunk size.

A large chunk size offers several important advantages. First, it reduces clients' need to interact with the master because reads and writes on the same chunk require only one initial request to the master for chunk location information. The reduction is especially significant for our workloads because applications mostly read and write large files sequentially. Even for small random reads, the client can comfortably cache all the chunk location information for a multi-TB working set. Second, since on a large chunk, a client is more likely to perform many operations on a given chunk, it can reduce network overhead by keeping a persistent TCP connection to the chunkserver over an extended period of time. Third, it reduces the size of the metadata stored on the master. This allows us to keep the metadata in memory, which in turn brings other advantages that we will discuss in Section 2.6.1.

On the other hand, a large chunk size, even with lazy space allocation, has its disadvantages. A small file consists of a small number of chunks, perhaps just one. The chunkservers storing those chunks may become hot spots if many clients are accessing the same file. In practice, hot spots have not been a major issue because our applications mostly read large multi-chunk files sequentially.

However, hot spots did develop when GFS was first used by a batch-queue system: an executable was written to GFS as a single-chunk file and then started on hundreds of machines at the same time. The few chunkservers storing this executable were overloaded by hundreds of simultaneous requests. We fixed this problem by storing such executables with a higher replication factor and by making the batch-queue system stagger application start times. A potential long-term solution is to allow clients to read data from other clients in such situations.

2.6 Metadata

The master stores three major types of metadata: the file and chunk namespaces, the mapping from files to chunks, and the locations of each chunk's replicas. All metadata is kept in the master's memory. The first two types (namespaces and file-to-chunk mapping) are also kept persistent by logging mutations to an *operation log* stored on the master's local disk and replicated on remote machines. Using a log allows us to update the master state simply, reliably, and without risking inconsistencies in the event of a master crash. The master does not store chunk location information persistently. Instead, it asks each chunkserver about its chunks at master startup and whenever a chunkserver joins the cluster.

2.6.1 In-Memory Data Structures

Since metadata is stored in memory, master operations are fast. Furthermore, it is easy and efficient for the master to periodically scan through its entire state in the background. This periodic scanning is used to implement chunk garbage collection, re-replication in the presence of chunkserver failures, and chunk migration to balance load and disk space

usage across chunkservers. Sections 4.3 and 4.4 will discuss these activities further.

One potential concern for this memory-only approach is that the number of chunks and hence the capacity of the whole system is limited by how much memory the master has. This is not a serious limitation in practice. The master maintains less than 64 bytes of metadata for each 64 MB chunk. Most chunks are full because most files contain many chunks, only the last of which may be partially filled. Similarly, the file namespace data typically requires less then 64 bytes per file because it stores file names compactly using prefix compression.

If necessary to support even larger file systems, the cost of adding extra memory to the master is a small price to pay for the simplicity, reliability, performance, and flexibility we gain by storing the metadata in memory.

2.6.2 Chunk Locations

The master does not keep a persistent record of which chunkservers have a replica of a given chunk. It simply polls chunkservers for that information at startup. The master can keep itself up-to-date thereafter because it controls all chunk placement and monitors chunkserver status with regular *HeartBeat* messages.

We initially attempted to keep chunk location information persistently at the master, but we decided that it was much simpler to request the data from chunkservers at startup, and periodically thereafter. This eliminated the problem of keeping the master and chunkservers in sync as chunkservers join and leave the cluster, change names, fail, restart, and so on. In a cluster with hundreds of servers, these events happen all too often.

Another way to understand this design decision is to realize that a chunkserver has the final word over what chunks it does or does not have on its own disks. There is no point in trying to maintain a consistent view of this information on the master because errors on a chunkserver may cause chunks to vanish spontaneously (e.g., a disk may go bad and be disabled) or an operator may rename a chunkserver.

2.6.3 Operation Log

The operation log contains a historical record of critical metadata changes. It is central to GFS. Not only is it the only persistent record of metadata, but it also serves as a logical time line that defines the order of concurrent operations. Files and chunks, as well as their versions (see Section 4.5), are all uniquely and eternally identified by the logical times at which they were created.

Since the operation log is critical, we must store it reliably and not make changes visible to clients until metadata changes are made persistent. Otherwise, we effectively lose the whole file system or recent client operations even if the chunks themselves survive. Therefore, we replicate it on multiple remote machines and respond to a client operation only after flushing the corresponding log record to disk both locally and remotely. The master batches several log records together before flushing thereby reducing the impact of flushing and replication on overall system throughput.

The master recovers its file system state by replaying the operation log. To minimize startup time, we must keep the log small. The master checkpoints its state whenever the log grows beyond a certain size so that it can recover by loading the latest checkpoint from local disk and replaying only the limited number of log records after that. The checkpoint is in a compact B-tree like form that can be directly mapped into memory and used for namespace lookup without extra parsing. This further speeds up recovery and improves availability.

Because building a checkpoint can take a while, the master's internal state is structured in such a way that a new checkpoint can be created without delaying incoming mutations. The master switches to a new log file and creates the new checkpoint in a separate thread. The new checkpoint includes all mutations before the switch. It can be created in a minute or so for a cluster with a few million files. When completed, it is written to disk both locally and remotely.

Recovery needs only the latest complete checkpoint and subsequent log files. Older checkpoints and log files can be freely deleted, though we keep a few around to guard against catastrophes. A failure during checkpointing does not affect correctness because the recovery code detects and skips incomplete checkpoints.

	Write	Record Append
Serial success	*defined*	*defined* interspersed with *inconsistent*
Concurrent successes	*consistent* but *undefined*	
Failure	*inconsistent*	

Table 1: File Region State After Mutation

2.7 Consistency Model

GFS has a relaxed consistency model that supports our highly distributed applications well but remains relatively simple and efficient to implement. We now discuss GFS's guarantees and what they mean to applications. We also highlight how GFS maintains these guarantees but leave the details to other parts of the paper.

2.7.1 Guarantees by GFS

File namespace mutations (e.g., file creation) are atomic. They are handled exclusively by the master: namespace locking guarantees atomicity and correctness (Section 4.1); the master's operation log defines a global total order of these operations (Section 2.6.3).

The state of a file region after a data mutation depends on the type of mutation, whether it succeeds or fails, and whether there are concurrent mutations. Table 1 summarizes the result. A file region is *consistent* if all clients will always see the same data, regardless of which replicas they read from. A region is *defined* after a file data mutation if it is consistent and clients will see what the mutation writes in its entirety. When a mutation succeeds without interference from concurrent writers, the affected region is defined (and by implication consistent): all clients will always see what the mutation has written. Concurrent successful mutations leave the region undefined but consistent: all clients see the same data, but it may not reflect what any one mutation has written. Typically, it consists of mingled fragments from multiple mutations. A failed mutation makes the region inconsistent (hence also undefined): different clients may see different data at different times. We describe below how our applications can distinguish defined regions from undefined

regions. The applications do not need to further distinguish between different kinds of undefined regions.

Data mutations may be *writes* or *record appends*. A write causes data to be written at an application-specified file offset. A record append causes data (the "record") to be appended *atomically at least once* even in the presence of concurrent mutations, but at an offset of GFS's choosing (Section 3.3). (In contrast, a "regular" append is merely a write at an offset that the client believes to be the current end of file.) The offset is returned to the client and marks the beginning of a defined region that contains the record. In addition, GFS may insert padding or record duplicates in between. They occupy regions considered to be inconsistent and are typically dwarfed by the amount of user data.

After a sequence of successful mutations, the mutated file region is guaranteed to be defined and contain the data written by the last mutation. GFS achieves this by (a) applying mutations to a chunk in the same order on all its replicas (Section 3.1), and (b) using chunk version numbers to detect any replica that has become stale because it has missed mutations while its chunkserver was down (Section 4.5). Stale replicas will never be involved in a mutation or given to clients asking the master for chunk locations. They are garbage collected at the earliest opportunity.

Since clients cache chunk locations, they may read from a stale replica before that information is refreshed. This window is limited by the cache entry's timeout and the next open of the file, which purges from the cache all chunk information for that file. Moreover, as most of our files are append-only, a stale replica usually returns a premature end of chunk rather than outdated data. When a reader retries and contacts the master, it will immediately get current chunk locations.

Long after a successful mutation, component failures can of course still corrupt or destroy data. GFS identifies failed chunkservers by regular handshakes between master and all chunkservers and detects data corruption by checksumming (Section 5.2). Once a problem surfaces, the data is restored from valid replicas as soon as possible (Section 4.3). A chunk is lost irreversibly only if all its replicas are lost before GFS can react, typically within minutes. Even in this case, it becomes unavailable, not corrupted: applications receive clear errors rather than corrupt data.

2.7.2 Implications for Applications

GFS applications can accommodate the relaxed consistency model with a few simple techniques already needed for other purposes: relying on appends rather than overwrites, checkpointing, and writing self-validating, self-identifying records.

Practically all our applications mutate files by appending rather than overwriting. In one typical use, a writer generates a file from beginning to end. It atomically renames the file to a permanent name after writing all the data, or periodically checkpoints how much has been successfully written. Checkpoints may also include application-level checksums. Readers verify and process only the file region up to the last checkpoint, which is known to be in the defined state. Regardless of consistency and concurrency issues, this approach has served us well. Appending is far more efficient and more resilient to application failures than random writes. Checkpointing allows writers to restart incrementally and keeps readers from processing successfully written file data that is still incomplete from the application's perspective.

In the other typical use, many writers concurrently append to a file for merged results or as a producer-consumer queue. Record append's append-at-least-once semantics preserves each writer's output. Readers deal with the occasional padding and duplicates as follows. Each record prepared by the writer contains extra information like checksums so that its validity can be verified. A reader can identify and discard extra padding and record fragments using the checksums. If it cannot tolerate the occasional duplicates (e.g., if they would trigger non-idempotent operations), it can filter them out using unique identifiers in the records, which are often needed anyway to name corresponding application entities such as web documents. These functionalities for record I/O (except duplicate removal) are in library code shared by our applications and applicable to other file interface implementations at Google. With that, the same sequence of records, plus rare duplicates, is always delivered to the record reader.

3. SYSTEM INTERACTIONS

We designed the system to minimize the master's involvement in all operations. With that background, we now describe how the client, master, and chunkservers interact to implement data mutations, atomic record append, and snapshot.

3.1 Leases and Mutation Order

A mutation is an operation that changes the contents or metadata of a chunk such as a write or an append operation. Each mutation is performed at all the chunk's replicas. We use leases to maintain a consistent mutation order across replicas. The master grants a chunk lease to one of the replicas, which we call the *primary*. The primary picks a serial order for all mutations to the chunk. All replicas follow this order when applying mutations. Thus, the global mutation order is defined first by the lease grant order chosen by the master, and within a lease by the serial numbers assigned by the primary.

The lease mechanism is designed to minimize management overhead at the master. A lease has an initial timeout of 60 seconds. However, as long as the chunk is being mutated, the primary can request and typically receive extensions from the master indefinitely. These extension requests and grants are piggybacked on the *HeartBeat* messages regularly exchanged between the master and all chunkservers. The master may sometimes try to revoke a lease before it expires (e.g., when the master wants to disable mutations on a file that is being renamed). Even if the master loses communication with a primary, it can safely grant a new lease to another replica after the old lease expires.

In Figure 2, we illustrate this process by following the control flow of a write through these numbered steps.

1. The client asks the master which chunkserver holds the current lease for the chunk and the locations of the other replicas. If no one has a lease, the master grants one to a replica it chooses (not shown).
2. The master replies with the identity of the primary and the locations of the other (*secondary*) replicas. The client caches this data for future mutations. It needs to contact the master again only when the primary

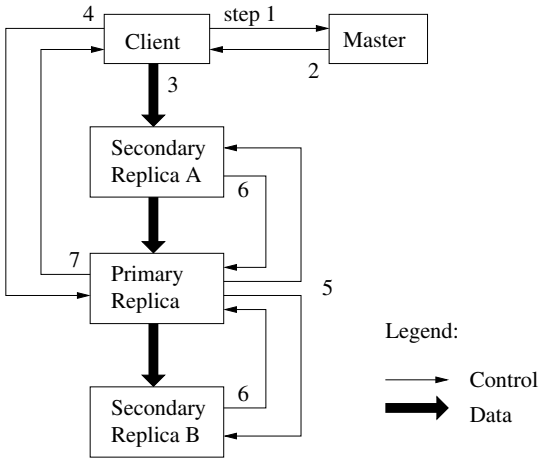


Figure 2: Write Control and Data Flow

becomes unreachable or replies that it no longer holds a lease.

3. The client pushes the data to all the replicas. A client can do so in any order. Each chunkserver will store the data in an internal LRU buffer cache until the data is used or aged out. By decoupling the data flow from the control flow, we can improve performance by scheduling the expensive data flow based on the network topology regardless of which chunkserver is the primary. Section 3.2 discusses this further.
4. Once all the replicas have acknowledged receiving the data, the client sends a write request to the primary. The request identifies the data pushed earlier to all of the replicas. The primary assigns consecutive serial numbers to all the mutations it receives, possibly from multiple clients, which provides the necessary serialization. It applies the mutation to its own local state in serial number order.
5. The primary forwards the write request to all secondary replicas. Each secondary replica applies mutations in the same serial number order assigned by the primary.
6. The secondaries all reply to the primary indicating that they have completed the operation.
7. The primary replies to the client. Any errors encountered at any of the replicas are reported to the client. In case of errors, the write may have succeeded at the primary and an arbitrary subset of the secondary replicas. (If it had failed at the primary, it would not have been assigned a serial number and forwarded.) The client request is considered to have failed, and the modified region is left in an inconsistent state. Our client code handles such errors by retrying the failed mutation. It will make a few attempts at steps (3) through (7) before falling back to a retry from the beginning of the write.

If a write by the application is large or straddles a chunk boundary, GFS client code breaks it down into multiple write operations. They all follow the control flow described above but may be interleaved with and overwritten by concurrent operations from other clients. Therefore, the shared file region may end up containing fragments from different clients, although the replicas will be identical because the individual operations are completed successfully in the same order on all replicas. This leaves the file region in consistent but undefined state as noted in Section 2.7.

3.2 Data Flow

We decouple the flow of data from the flow of control to use the network efficiently. While control flows from the client to the primary and then to all secondaries, data is pushed linearly along a carefully picked chain of chunkservers in a pipelined fashion. Our goals are to fully utilize each machine's network bandwidth, avoid network bottlenecks and high-latency links, and minimize the latency to push through all the data.

To fully utilize each machine's network bandwidth, the data is pushed linearly along a chain of chunkservers rather than distributed in some other topology (e.g., tree). Thus, each machine's full outbound bandwidth is used to transfer the data as fast as possible rather than divided among multiple recipients.

To avoid network bottlenecks and high-latency links (e.g., inter-switch links are often both) as much as possible, each machine forwards the data to the "closest" machine in the network topology that has not received it. Suppose the client is pushing data to chunkservers S1 through S4. It sends the data to the closest chunkserver, say S1. S1 forwards it to the closest chunkserver S2 through S4 closest to S1, say S2. Similarly, S2 forwards it to S3 or S4, whichever is closer to S2, and so on. Our network topology is simple enough that "distances" can be accurately estimated from IP addresses.

Finally, we minimize latency by pipelining the data transfer over TCP connections. Once a chunkserver receives some data, it starts forwarding immediately. Pipelining is especially helpful to us because we use a switched network with full-duplex links. Sending the data immediately does not reduce the receive rate. Without network congestion, the ideal elapsed time for transferring B bytes to R replicas is $B/T + RL$ where T is the network throughput and L is latency to transfer bytes between two machines. Our network links are typically 100 Mbps (T), and L is far below 1 ms. Therefore, 1 MB can ideally be distributed in about 80 ms.

3.3 Atomic Record Appends

GFS provides an atomic append operation called *record append*. In a traditional write, the client specifies the offset at which data is to be written. Concurrent writes to the same region are not serializable: the region may end up containing data fragments from multiple clients. In a record append, however, the client specifies only the data. GFS appends it to the file at least once atomically (i.e., as one continuous sequence of bytes) at an offset of GFS's choosing and returns that offset to the client. This is similar to writing to a file opened in `O_APPEND` mode in Unix without the race conditions when multiple writers do so concurrently.

Record append is heavily used by our distributed applications in which many clients on different machines append to the same file concurrently. Clients would need additional complicated and expensive synchronization, for example through a distributed lock manager, if they do so with traditional writes. In our workloads, such files often

serve as multiple-producer/single-consumer queues or contain merged results from many different clients.

Record append is a kind of mutation and follows the control flow in Section 3.1 with only a little extra logic at the primary. The client pushes the data to all replicas of the last chunk of the file Then, it sends its request to the primary. The primary checks to see if appending the record to the current chunk would cause the chunk to exceed the maximum size (64 MB). If so, it pads the chunk to the maximum size, tells secondaries to do the same, and replies to the client indicating that the operation should be retried on the next chunk. (Record append is restricted to be at most one-fourth of the maximum chunk size to keep worst-case fragmentation at an acceptable level.) If the record fits within the maximum size, which is the common case, the primary appends the data to its replica, tells the secondaries to write the data at the exact offset where it has, and finally replies success to the client.

If a record append fails at any replica, the client retries the operation. As a result, replicas of the same chunk may contain different data possibly including duplicates of the same record in whole or in part. GFS does not guarantee that all replicas are bytewise identical. It only guarantees that the data is written at least once as an atomic unit. This property follows readily from the simple observation that for the operation to report success, the data must have been written at the same offset on all replicas of some chunk. Furthermore, after this, all replicas are at least as long as the end of record and therefore any future record will be assigned a higher offset or a different chunk even if a different replica later becomes the primary. In terms of our consistency guarantees, the regions in which successful record append operations have written their data are defined (hence consistent), whereas intervening regions are inconsistent (hence undefined). Our applications can deal with inconsistent regions as we discussed in Section 2.7.2.

3.4 Snapshot

The snapshot operation makes a copy of a file or a directory tree (the "source") almost instantaneously, while minimizing any interruptions of ongoing mutations. Our users use it to quickly create branch copies of huge data sets (and often copies of those copies, recursively), or to checkpoint the current state before experimenting with changes that can later be committed or rolled back easily.

Like AFS [5], we use standard copy-on-write techniques to implement snapshots. When the master receives a snapshot request, it first revokes any outstanding leases on the chunks in the files it is about to snapshot. This ensures that any subsequent writes to these chunks will require an interaction with the master to find the lease holder. This will give the master an opportunity to create a new copy of the chunk first.

After the leases have been revoked or have expired, the master logs the operation to disk. It then applies this log record to its in-memory state by duplicating the metadata for the source file or directory tree. The newly created snapshot files point to the same chunks as the source files.

The first time a client wants to write to a chunk C after the snapshot operation, it sends a request to the master to find the current lease holder. The master notices that the reference count for chunk C is greater than one. It defers replying to the client request and instead picks a new chunk handle C'. It then asks each chunkserver that has a current replica of C to create a new chunk called C'. By creating the new chunk on the same chunkservers as the original, we ensure that the data can be copied locally, not over the network (our disks are about three times as fast as our 100 Mb Ethernet links). From this point, request handling is no different from that for any chunk: the master grants one of the replicas a lease on the new chunk C' and replies to the client, which can write the chunk normally, not knowing that it has just been created from an existing chunk.

4. MASTER OPERATION

The master executes all namespace operations. In addition, it manages chunk replicas throughout the system: it makes placement decisions, creates new chunks and hence replicas, and coordinates various system-wide activities to keep chunks fully replicated, to balance load across all the chunkservers, and to reclaim unused storage. We now discuss each of these topics.

4.1 Namespace Management and Locking

Many master operations can take a long time: for example, a snapshot operation has to revoke chunkserver leases on all chunks covered by the snapshot. We do not want to delay other master operations while they are running. Therefore, we allow multiple operations to be active and use locks over regions of the namespace to ensure proper serialization.

Unlike many traditional file systems, GFS does not have a per-directory data structure that lists all the files in that directory. Nor does it support aliases for the same file or directory (i.e, hard or symbolic links in Unix terms). GFS logically represents its namespace as a lookup table mapping full pathnames to metadata. With prefix compression, this table can be efficiently represented in memory. Each node in the namespace tree (either an absolute file name or an absolute directory name) has an associated read-write lock.

Each master operation acquires a set of locks before it runs. Typically, if it involves `/d1/d2/.../dn/leaf`, it will acquire read-locks on the directory names `/d1`, `/d1/d2`, ..., `/d1/d2/.../dn`, and either a read lock or a write lock on the full pathname `/d1/d2/.../dn/leaf`. Note that `leaf` may be a file or directory depending on the operation.

We now illustrate how this locking mechanism can prevent a file `/home/user/foo` from being created while `/home/user` is being snapshotted to `/save/user`. The snapshot operation acquires read locks on `/home` and `/save`, and write locks on `/home/user` and `/save/user`. The file creation acquires read locks on `/home` and `/home/user`, and a write lock on `/home/user/foo`. The two operations will be serialized properly because they try to obtain conflicting locks on `/home/user`. File creation does not require a write lock on the parent directory because there is no "directory", or *inode*-like, data structure to be protected from modification. The read lock on the name is sufficient to protect the parent directory from deletion.

One nice property of this locking scheme is that it allows concurrent mutations in the same directory. For example, multiple file creations can be executed concurrently in the same directory: each acquires a read lock on the directory name and a write lock on the file name. The read lock on the directory name suffices to prevent the directory from being deleted, renamed, or snapshotted. The write locks on

file names serialize attempts to create a file with the same name twice.

Since the namespace can have many nodes, read-write lock objects are allocated lazily and deleted once they are not in use. Also, locks are acquired in a consistent total order to prevent deadlock: they are first ordered by level in the namespace tree and lexicographically within the same level.

4.2 Replica Placement

A GFS cluster is highly distributed at more levels than one. It typically has hundreds of chunkservers spread across many machine racks. These chunkservers in turn may be accessed from hundreds of clients from the same or different racks. Communication between two machines on different racks may cross one or more network switches. Additionally, bandwidth into or out of a rack may be less than the aggregate bandwidth of all the machines within the rack. Multi-level distribution presents a unique challenge to distribute data for scalability, reliability, and availability.

The chunk replica placement policy serves two purposes: maximize data reliability and availability, and maximize network bandwidth utilization. For both, it is not enough to spread replicas across machines, which only guards against disk or machine failures and fully utilizes each machine's network bandwidth. We must also spread chunk replicas across racks. This ensures that some replicas of a chunk will survive and remain available even if an entire rack is damaged or offline (for example, due to failure of a shared resource like a network switch or power circuit). It also means that traffic, especially reads, for a chunk can exploit the aggregate bandwidth of multiple racks. On the other hand, write traffic has to flow through multiple racks, a tradeoff we make willingly.

4.3 Creation, Re-replication, Rebalancing

Chunk replicas are created for three reasons: chunk creation, re-replication, and rebalancing.

When the master *creates* a chunk, it chooses where to place the initially empty replicas. It considers several factors. (1) We want to place new replicas on chunkservers with below-average disk space utilization. Over time this will equalize disk utilization across chunkservers. (2) We want to limit the number of "recent" creations on each chunkserver. Although creation itself is cheap, it reliably predicts imminent heavy write traffic because chunks are created when demanded by writes, and in our append-once-read-many workload they typically become practically read-only once they have been completely written. (3) As discussed above, we want to spread replicas of a chunk across racks.

The master *re-replicates* a chunk as soon as the number of available replicas falls below a user-specified goal. This could happen for various reasons: a chunkserver becomes unavailable, it reports that its replica may be corrupted, one of its disks is disabled because of errors, or the replication goal is increased. Each chunk that needs to be re-replicated is prioritized based on several factors. One is how far it is from its replication goal. For example, we give higher priority to a chunk that has lost two replicas than to a chunk that has lost only one. In addition, we prefer to first re-replicate chunks for live files as opposed to chunks that belong to recently deleted files (see Section 4.4). Finally, to minimize the impact of failures on running applications, we boost the priority of any chunk that is blocking client progress.

The master picks the highest priority chunk and "clones" it by instructing some chunkserver to copy the chunk data directly from an existing valid replica. The new replica is placed with goals similar to those for creation: equalizing disk space utilization, limiting active clone operations on any single chunkserver, and spreading replicas across racks. To keep cloning traffic from overwhelming client traffic, the master limits the numbers of active clone operations both for the cluster and for each chunkserver. Additionally, each chunkserver limits the amount of bandwidth it spends on each clone operation by throttling its read requests to the source chunkserver.

Finally, the master *rebalances* replicas periodically: it examines the current replica distribution and moves replicas for better disk space and load balancing. Also through this process, the master gradually fills up a new chunkserver rather than instantly swamps it with new chunks and the heavy write traffic that comes with them. The placement criteria for the new replica are similar to those discussed above. In addition, the master must also choose which existing replica to remove. In general, it prefers to remove those on chunkservers with below-average free space so as to equalize disk space usage.

4.4 Garbage Collection

After a file is deleted, GFS does not immediately reclaim the available physical storage. It does so only lazily during regular garbage collection at both the file and chunk levels. We find that this approach makes the system much simpler and more reliable.

4.4.1 Mechanism

When a file is deleted by the application, the master logs the deletion immediately just like other changes. However instead of reclaiming resources immediately, the file is just renamed to a hidden name that includes the deletion timestamp. During the master's regular scan of the file system namespace, it removes any such hidden files if they have existed for more than three days (the interval is configurable). Until then, the file can still be read under the new, special name and can be undeleted by renaming it back to normal. When the hidden file is removed from the namespace, its in-memory metadata is erased. This effectively severs its links to all its chunks.

In a similar regular scan of the chunk namespace, the master identifies orphaned chunks (i.e., those not reachable from any file) and erases the metadata for those chunks. In a *HeartBeat* message regularly exchanged with the master, each chunkserver reports a subset of the chunks it has, and the master replies with the identity of all chunks that are no longer present in the master's metadata. The chunkserver is free to delete its replicas of such chunks.

4.4.2 Discussion

Although distributed garbage collection is a hard problem that demands complicated solutions in the context of programming languages, it is quite simple in our case. We can easily identify all references to chunks: they are in the file-to-chunk mappings maintained exclusively by the master. We can also easily identify all the chunk replicas: they are Linux files under designated directories on each chunkserver. Any such replica not known to the master is "garbage."

The garbage collection approach to storage reclamation offers several advantages over eager deletion. First, it is simple and reliable in a large-scale distributed system where component failures are common. Chunk creation may succeed on some chunkservers but not others, leaving replicas that the master does not know exist. Replica deletion messages may be lost, and the master has to remember to resend them across failures, both its own and the chunkserver's. Garbage collection provides a uniform and dependable way to clean up any replicas not known to be useful. Second, it merges storage reclamation into the regular background activities of the master, such as the regular scans of namespaces and handshakes with chunkservers. Thus, it is done in batches and the cost is amortized. Moreover, it is done only when the master is relatively free. The master can respond more promptly to client requests that demand timely attention. Third, the delay in reclaiming storage provides a safety net against accidental, irreversible deletion.

In our experience, the main disadvantage is that the delay sometimes hinders user effort to fine tune usage when storage is tight. Applications that repeatedly create and delete temporary files may not be able to reuse the storage right away. We address these issues by expediting storage reclamation if a deleted file is explicitly deleted again. We also allow users to apply different replication and reclamation policies to different parts of the namespace. For example, users can specify that all the chunks in the files within some directory tree are to be stored without replication, and any deleted files are immediately and irrevocably removed from the file system state.

4.5 Stale Replica Detection

Chunk replicas may become stale if a chunkserver fails and misses mutations to the chunk while it is down. For each chunk, the master maintains a *chunk version number* to distinguish between up-to-date and stale replicas.

Whenever the master grants a new lease on a chunk, it increases the chunk version number and informs the up-to-date replicas. The master and these replicas all record the new version number in their persistent state. This occurs before any client is notified and therefore before it can start writing to the chunk. If another replica is currently unavailable, its chunk version number will not be advanced. The master will detect that this chunkserver has a stale replica when the chunkserver restarts and reports its set of chunks and their associated version numbers. If the master sees a version number greater than the one in its records, the master assumes that it failed when granting the lease and so takes the higher version to be up-to-date.

The master removes stale replicas in its regular garbage collection. Before that, it effectively considers a stale replica not to exist at all when it replies to client requests for chunk information. As another safeguard, the master includes the chunk version number when it informs clients which chunkserver holds a lease on a chunk or when it instructs a chunkserver to read the chunk from another chunkserver in a cloning operation. The client or the chunkserver verifies the version number when it performs the operation so that it is always accessing up-to-date data.

5. FAULT TOLERANCE AND DIAGNOSIS

One of our greatest challenges in designing the system is dealing with frequent component failures. The quality and quantity of components together make these problems more the norm than the exception: we cannot completely trust the machines, nor can we completely trust the disks. Component failures can result in an unavailable system or, worse, corrupted data. We discuss how we meet these challenges and the tools we have built into the system to diagnose problems when they inevitably occur.

5.1 High Availability

Among hundreds of servers in a GFS cluster, some are bound to be unavailable at any given time. We keep the overall system highly available with two simple yet effective strategies: fast recovery and replication.

5.1.1 Fast Recovery

Both the master and the chunkserver are designed to restore their state and start in seconds no matter how they terminated. In fact, we do not distinguish between normal and abnormal termination; servers are routinely shut down just by killing the process. Clients and other servers experience a minor hiccup as they time out on their outstanding requests, reconnect to the restarted server, and retry. Section 6.2.2 reports observed startup times.

5.1.2 Chunk Replication

As discussed earlier, each chunk is replicated on multiple chunkservers on different racks. Users can specify different replication levels for different parts of the file namespace. The default is three. The master clones existing replicas as needed to keep each chunk fully replicated as chunkservers go offline or detect corrupted replicas through checksum verification (see Section 5.2). Although replication has served us well, we are exploring other forms of cross-server redundancy such as parity or erasure codes for our increasing read-only storage requirements. We expect that it is challenging but manageable to implement these more complicated redundancy schemes in our very loosely coupled system because our traffic is dominated by appends and reads rather than small random writes.

5.1.3 Master Replication

The master state is replicated for reliability. Its operation log and checkpoints are replicated on multiple machines. A mutation to the state is considered committed only after its log record has been flushed to disk locally and on all master replicas. For simplicity, one master process remains in charge of all mutations as well as background activities such as garbage collection that change the system internally. When it fails, it can restart almost instantly. If its machine or disk fails, monitoring infrastructure outside GFS starts a new master process elsewhere with the replicated operation log. Clients use only the canonical name of the master (e.g. gfs-test), which is a DNS alias that can be changed if the master is relocated to another machine.

Moreover, "shadow" masters provide read-only access to the file system even when the primary master is down. They are shadows, not mirrors, in that they may lag the primary slightly, typically fractions of a second. They enhance read availability for files that are not being actively mutated or applications that do not mind getting slightly stale results. In fact, since file content is read from chunkservers, applications do not observe stale file content. What could be

stale within short windows is file metadata, like directory contents or access control information.

To keep itself informed, a shadow master reads a replica of the growing operation log and applies the same sequence of changes to its data structures exactly as the primary does. Like the primary, it polls chunkservers at startup (and infrequently thereafter) to locate chunk replicas and exchanges frequent handshake messages with them to monitor their status. It depends on the primary master only for replica location updates resulting from the primary's decisions to create and delete replicas.

5.2 Data Integrity

Each chunkserver uses checksumming to detect corruption of stored data. Given that a GFS cluster often has thousands of disks on hundreds of machines, it regularly experiences disk failures that cause data corruption or loss on both the read and write paths. (See Section 7 for one cause.) We can recover from corruption using other chunk replicas, but it would be impractical to detect corruption by comparing replicas across chunkservers. Moreover, divergent replicas may be legal: the semantics of GFS mutations, in particular atomic record append as discussed earlier, does not guarantee identical replicas. Therefore, each chunkserver must independently verify the integrity of its own copy by maintaining checksums.

A chunk is broken up into 64 KB blocks. Each has a corresponding 32 bit checksum. Like other metadata, checksums are kept in memory and stored persistently with logging, separate from user data.

For reads, the chunkserver verifies the checksum of data blocks that overlap the read range before returning any data to the requester, whether a client or another chunkserver. Therefore chunkservers will not propagate corruptions to other machines. If a block does not match the recorded checksum, the chunkserver returns an error to the requestor and reports the mismatch to the master. In response, the requestor will read from other replicas, while the master will clone the chunk from another replica. After a valid new replica is in place, the master instructs the chunkserver that reported the mismatch to delete its replica.

Checksumming has little effect on read performance for several reasons. Since most of our reads span at least a few blocks, we need to read and checksum only a relatively small amount of extra data for verification. GFS client code further reduces this overhead by trying to align reads at checksum block boundaries. Moreover, checksum lookups and comparison on the chunkserver are done without any I/O, and checksum calculation can often be overlapped with I/Os.

Checksum computation is heavily optimized for writes that append to the end of a chunk (as opposed to writes that overwrite existing data) because they are dominant in our workloads. We just incrementally update the checksum for the last partial checksum block, and compute new checksums for any brand new checksum blocks filled by the append. Even if the last partial checksum block is already corrupted and we fail to detect it now, the new checksum value will not match the stored data, and the corruption will be detected as usual when the block is next read.

In contrast, if a write overwrites an existing range of the chunk, we must read and verify the first and last blocks of the range being overwritten, then perform the write, and finally compute and record the new checksums. If we do not verify the first and last blocks before overwriting them partially, the new checksums may hide corruption that exists in the regions not being overwritten.

During idle periods, chunkservers can scan and verify the contents of inactive chunks. This allows us to detect corruption in chunks that are rarely read. Once the corruption is detected, the master can create a new uncorrupted replica and delete the corrupted replica. This prevents an inactive but corrupted chunk replica from fooling the master into thinking that it has enough valid replicas of a chunk.

5.3 Diagnostic Tools

Extensive and detailed diagnostic logging has helped immeasurably in problem isolation, debugging, and performance analysis, while incurring only a minimal cost. Without logs, it is hard to understand transient, non-repeatable interactions between machines. GFS servers generate diagnostic logs that record many significant events (such as chunkservers going up and down) and all RPC requests and replies. These diagnostic logs can be freely deleted without affecting the correctness of the system. However, we try to keep these logs around as far as space permits.

The RPC logs include the exact requests and responses sent on the wire, except for the file data being read or written. By matching requests with replies and collating RPC records on different machines, we can reconstruct the entire interaction history to diagnose a problem. The logs also serve as traces for load testing and performance analysis.

The performance impact of logging is minimal (and far outweighed by the benefits) because these logs are written sequentially and asynchronously. The most recent events are also kept in memory and available for continuous online monitoring.

6. MEASUREMENTS

In this section we present a few micro-benchmarks to illustrate the bottlenecks inherent in the GFS architecture and implementation, and also some numbers from real clusters in use at Google.

6.1 Micro-benchmarks

We measured performance on a GFS cluster consisting of one master, two master replicas, 16 chunkservers, and 16 clients. Note that this configuration was set up for ease of testing. Typical clusters have hundreds of chunkservers and hundreds of clients.

All the machines are configured with dual 1.4 GHz PIII processors, 2 GB of memory, two 80 GB 5400 rpm disks, and a 100 Mbps full-duplex Ethernet connection to an HP 2524 switch. All 19 GFS server machines are connected to one switch, and all 16 client machines to the other. The two switches are connected with a 1 Gbps link.

6.1.1 Reads

N clients read simultaneously from the file system. Each client reads a randomly selected 4 MB region from a 320 GB file set. This is repeated 256 times so that each client ends up reading 1 GB of data. The chunkservers taken together have only 32 GB of memory, so we expect at most a 10% hit rate in the Linux buffer cache. Our results should be close to cold cache results.

Figure 3(a) shows the aggregate read rate for N clients and its theoretical limit. The limit peaks at an aggregate of 125 MB/s when the 1 Gbps link between the two switches is saturated, or 12.5 MB/s per client when its 100 Mbps network interface gets saturated, whichever applies. The observed read rate is 10 MB/s, or 80% of the per-client limit, when just one client is reading. The aggregate read rate reaches 94 MB/s, about 75% of the 125 MB/s link limit, for 16 readers, or 6 MB/s per client. The efficiency drops from 80% to 75% because as the number of readers increases, so does the probability that multiple readers simultaneously read from the same chunkserver.

6.1.2 Writes

N clients write simultaneously to N distinct files. Each client writes 1 GB of data to a new file in a series of 1 MB writes. The aggregate write rate and its theoretical limit are shown in Figure 3(b). The limit plateaus at 67 MB/s because we need to write each byte to 3 of the 16 chunkservers, each with a 12.5 MB/s input connection.

The write rate for one client is 6.3 MB/s, about half of the limit. The main culprit for this is our network stack. It does not interact very well with the pipelining scheme we use for pushing data to chunk replicas. Delays in propagating data from one replica to another reduce the overall write rate.

Aggregate write rate reaches 35 MB/s for 16 clients (or 2.2 MB/s per client), about half the theoretical limit. As in the case of reads, it becomes more likely that multiple clients write concurrently to the same chunkserver as the number of clients increases. Moreover, collision is more likely for 16 writers than for 16 readers because each write involves three different replicas.

Writes are slower than we would like. In practice this has not been a major problem because even though it increases the latencies as seen by individual clients, it does not significantly affect the aggregate write bandwidth delivered by the system to a large number of clients.

6.1.3 Record Appends

Figure 3(c) shows record append performance. N clients append simultaneously to a single file. Performance is limited by the network bandwidth of the chunkservers that store the last chunk of the file, independent of the number of clients. It starts at 6.0 MB/s for one client and drops to 4.8 MB/s for 16 clients, mostly due to congestion and variances in network transfer rates seen by different clients.

Our applications tend to produce multiple such files concurrently. In other words, N clients append to M shared files simultaneously where both N and M are in the dozens or hundreds. Therefore, the chunkserver network congestion in our experiment is not a significant issue in practice because a client can make progress on writing one file while the chunkservers for another file are busy.

6.2 Real World Clusters

We now examine two clusters in use within Google that are representative of several others like them. Cluster A is used regularly for research and development by over a hundred engineers. A typical task is initiated by a human user and runs up to several hours. It reads through a few MBs to a few TBs of data, transforms or analyzes the data, and writes the results back to the cluster. Cluster B is primarily used for production data processing. The tasks last much longer and continuously generate and process multi-TB data sets with only occasional human intervention. In both cases, a single "task" consists of many processes on many machines reading and writing many files simultaneously.

Cluster	A	B
Chunkservers	342	227
Available disk space	72 TB	180 TB
Used disk space	55 TB	155 TB
Number of Files	735 k	737 k
Number of Dead files	22 k	232 k
Number of Chunks	992 k	1550 k
Metadata at chunkservers	13 GB	21 GB
Metadata at master	48 MB	60 MB

Table 2: Characteristics of two GFS clusters

6.2.1 Storage

As shown by the first five entries in the table, both clusters have hundreds of chunkservers, support many TBs of disk space, and are fairly but not completely full. "Used space" includes all chunk replicas. Virtually all files are replicated three times. Therefore, the clusters store 18 TB and 52 TB of file data respectively.

The two clusters have similar numbers of files, though B has a larger proportion of dead files, namely files which were deleted or replaced by a new version but whose storage have not yet been reclaimed. It also has more chunks because its files tend to be larger.

6.2.2 Metadata

The chunkservers in aggregate store tens of GBs of metadata, mostly the checksums for 64 KB blocks of user data. The only other metadata kept at the chunkservers is the chunk version number discussed in Section 4.5.

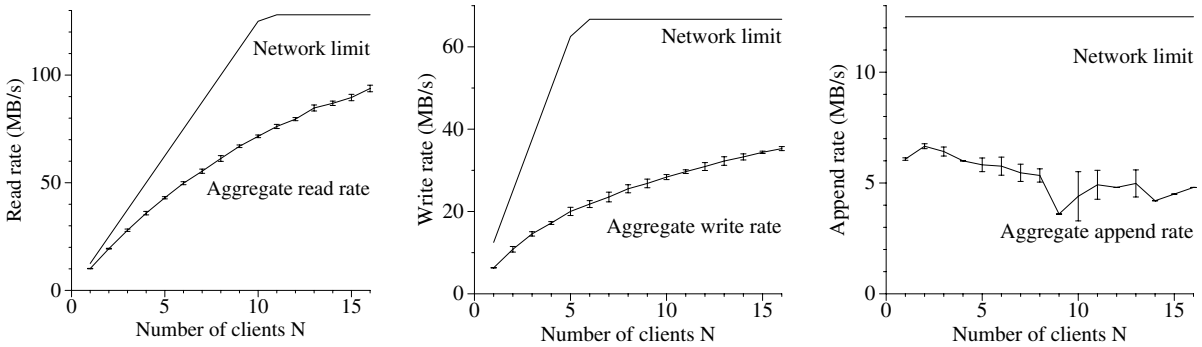
The metadata kept at the master is much smaller, only tens of MBs, or about 100 bytes per file on average. This agrees with our assumption that the size of the master's memory does not limit the system's capacity in practice. Most of the per-file metadata is the file names stored in a prefix-compressed form. Other metadata includes file ownership and permissions, mapping from files to chunks, and each chunk's current version. In addition, for each chunk we store the current replica locations and a reference count for implementing copy-on-write.

Each individual server, both chunkservers and the master, has only 50 to 100 MB of metadata. Therefore recovery is fast: it takes only a few seconds to read this metadata from disk before the server is able to answer queries. However, the master is somewhat hobbled for a period – typically 30 to 60 seconds – until it has fetched chunk location information from all chunkservers.

6.2.3 Read and Write Rates

Table 3 shows read and write rates for various time periods. Both clusters had been up for about one week when these measurements were taken. (The clusters had been restarted recently to upgrade to a new version of GFS.)

The average write rate was less than 30 MB/s since the restart. When we took these measurements, B was in the middle of a burst of write activity generating about 100 MB/s of data, which produced a 300 MB/s network load because writes are propagated to three replicas.



(a) Reads (b) Writes (c) Record appends

Figure 3: Aggregate Throughputs. Top curves show theoretical limits imposed by our network topology. Bottom curves show measured throughputs. They have error bars that show 95% confidence intervals, which are illegible in some cases because of low variance in measurements.

Cluster	A	B
Read rate (last minute)	583 MB/s	380 MB/s
Read rate (last hour)	562 MB/s	384 MB/s
Read rate (since restart)	589 MB/s	49 MB/s
Write rate (last minute)	1 MB/s	101 MB/s
Write rate (last hour)	2 MB/s	117 MB/s
Write rate (since restart)	25 MB/s	13 MB/s
Master ops (last minute)	325 Ops/s	533 Ops/s
Master ops (last hour)	381 Ops/s	518 Ops/s
Master ops (since restart)	202 Ops/s	347 Ops/s

Table 3: Performance Metrics for Two GFS Clusters

The read rates were much higher than the write rates. The total workload consists of more reads than writes as we have assumed. Both clusters were in the middle of heavy read activity. In particular, A had been sustaining a read rate of 580 MB/s for the preceding week. Its network configuration can support 750 MB/s, so it was using its resources efficiently. Cluster B can support peak read rates of 1300 MB/s, but its applications were using just 380 MB/s.

6.2.4 Master Load

Table 3 also shows that the rate of operations sent to the master was around 200 to 500 operations per second. The master can easily keep up with this rate, and therefore is not a bottleneck for these workloads.

In an earlier version of GFS, the master was occasionally a bottleneck for some workloads. It spent most of its time sequentially scanning through large directories (which contained hundreds of thousands of files) looking for particular files. We have since changed the master data structures to allow efficient binary searches through the namespace. It can now easily support many thousands of file accesses per second. If necessary, we could speed it up further by placing name lookup caches in front of the namespace data structures.

6.2.5 Recovery Time

After a chunkserver fails, some chunks will become under-replicated and must be cloned to restore their replication levels. The time it takes to restore all such chunks depends on the amount of resources. In one experiment, we killed a single chunkserver in cluster B. The chunkserver had about 15,000 chunks containing 600 GB of data. To limit the impact on running applications and provide leeway for scheduling decisions, our default parameters limit this cluster to 91 concurrent clonings (40% of the number of chunkservers) where each clone operation is allowed to consume at most 6.25 MB/s (50 Mbps). All chunks were restored in 23.2 minutes, at an effective replication rate of 440 MB/s.

In another experiment, we killed two chunkservers each with roughly 16,000 chunks and 660 GB of data. This double failure reduced 266 chunks to having a single replica. These 266 chunks were cloned at a higher priority, and were all restored to at least 2x replication within 2 minutes, thus putting the cluster in a state where it could tolerate another chunkserver failure without data loss.

6.3 Workload Breakdown

In this section, we present a detailed breakdown of the workloads on two GFS clusters comparable but not identical to those in Section 6.2. Cluster X is for research and development while cluster Y is for production data processing.

6.3.1 Methodology and Caveats

These results include only client originated requests so that they reflect the workload generated by our applications for the file system as a whole. They do not include inter-server requests to carry out client requests or internal background activities, such as forwarded writes or rebalancing.

Statistics on I/O operations are based on information heuristically reconstructed from actual RPC requests logged by GFS servers. For example, GFS client code may break a read into multiple RPCs to increase parallelism, from which we infer the original read. Since our access patterns are highly stylized, we expect any error to be in the noise. Explicit logging by applications might have provided slightly more accurate data, but it is logistically impossible to recompile and restart thousands of running clients to do so and cumbersome to collect the results from as many machines.

One should be careful not to overly generalize from our workload. Since Google completely controls both GFS and its applications, the applications tend to be tuned for GFS, and conversely GFS is designed for these applications. Such mutual influence may also exist between general applications

Operation	Read		Write		Record Append	
Cluster	X	Y	X	Y	X	Y
0K	0.4	2.6	0	0	0	0
1B..1K	0.1	4.1	6.6	4.9	0.2	9.2
1K..8K	65.2	38.5	0.4	1.0	18.9	15.2
8K..64K	29.9	45.1	17.8	43.0	78.0	2.8
64K..128K	0.1	0.7	2.3	1.9	< .1	4.3
128K..256K	0.2	0.3	31.6	0.4	< .1	10.6
256K..512K	0.1	0.1	4.2	7.7	< .1	31.2
512K..1M	3.9	6.9	35.5	28.7	2.2	25.5
1M..inf	0.1	1.8	1.5	12.3	0.7	2.2

Table 4: Operations Breakdown by Size (%). For reads, the size is the amount of data actually read and transferred, rather than the amount requested.

and file systems, but the effect is likely more pronounced in our case.

6.3.2 Chunkserver Workload

Table 4 shows the distribution of operations by size. Read sizes exhibit a bimodal distribution. The small reads (under 64 KB) come from seek-intensive clients that look up small pieces of data within huge files. The large reads (over 512 KB) come from long sequential reads through entire files.

A significant number of reads return no data at all in cluster Y. Our applications, especially those in the production systems, often use files as producer-consumer queues. Producers append concurrently to a file while a consumer reads the end of file. Occasionally, no data is returned when the consumer outpaces the producers. Cluster X shows this less often because it is usually used for short-lived data analysis tasks rather than long-lived distributed applications.

Write sizes also exhibit a bimodal distribution. The large writes (over 256 KB) typically result from significant buffering within the writers. Writers that buffer less data, checkpoint or synchronize more often, or simply generate less data account for the smaller writes (under 64 KB).

As for record appends, cluster Y sees a much higher percentage of large record appends than cluster X does because our production systems, which use cluster Y, are more aggressively tuned for GFS.

Table 5 shows the total amount of data transferred in operations of various sizes. For all kinds of operations, the larger operations (over 256 KB) generally account for most of the bytes transferred. Small reads (under 64 KB) do transfer a small but significant portion of the read data because of the random seek workload.

6.3.3 Appends versus Writes

Record appends are heavily used especially in our production systems. For cluster X, the ratio of writes to record appends is 108:1 by bytes transferred and 8:1 by operation counts. For cluster Y, used by the production systems, the ratios are 3.7:1 and 2.5:1 respectively. Moreover, these ratios suggest that for both clusters record appends tend to be larger than writes. For cluster X, however, the overall usage of record append during the measured period is fairly low and so the results are likely skewed by one or two applications with particular buffer size choices.

As expected, our data mutation workload is dominated by appending rather than overwriting. We measured the amount of data overwritten on primary replicas. This approximates the case where a client deliberately overwrites previous written data rather than appends new data. For cluster X, overwriting accounts for under 0.0001% of bytes mutated and under 0.0003% of mutation operations. For cluster Y, the ratios are both 0.05%. Although this is minute, it is still higher than we expected. It turns out that most of these overwrites came from client retries due to errors or timeouts. They are not part of the workload *per se* but a consequence of the retry mechanism.

Operation	Read		Write		Record Append	
Cluster	X	Y	X	Y	X	Y
1B..1K	< .1	< .1	< .1	< .1	< .1	< .1
1K..8K	13.8	3.9	< .1	< .1	< .1	0.1
8K..64K	11.4	9.3	2.4	5.9	2.3	0.3
64K..128K	0.3	0.7	0.3	0.3	22.7	1.2
128K..256K	0.8	0.6	16.5	0.2	< .1	5.8
256K..512K	1.4	0.3	3.4	7.7	< .1	38.4
512K..1M	65.9	55.1	74.1	58.0	.1	46.8
1M..inf	6.4	30.1	3.3	28.0	53.9	7.4

Table 5: Bytes Transferred Breakdown by Operation Size (%). For reads, the size is the amount of data actually read and transferred, rather than the amount requested. The two may differ if the read attempts to read beyond end of file, which by design is not uncommon in our workloads.

Cluster	X	Y
Open	26.1	16.3
Delete	0.7	1.5
FindLocation	64.3	65.8
FindLeaseHolder	7.8	13.4
FindMatchingFiles	0.6	2.2
All other combined	0.5	0.8

Table 6: Master Requests Breakdown by Type (%)

6.3.4 Master Workload

Table 6 shows the breakdown by type of requests to the master. Most requests ask for chunk locations (*FindLocation*) for reads and lease holder information (*FindLeaseLocker*) for data mutations.

Clusters X and Y see significantly different numbers of *Delete* requests because cluster Y stores production data sets that are regularly regenerated and replaced with newer versions. Some of this difference is further hidden in the difference in *Open* requests because an old version of a file may be implicitly deleted by being opened for write from scratch (mode "w" in Unix open terminology).

FindMatchingFiles is a pattern matching request that supports "ls" and similar file system operations. Unlike other requests for the master, it may process a large part of the namespace and so may be expensive. Cluster Y sees it much more often because automated data processing tasks tend to examine parts of the file system to understand global application state. In contrast, cluster X's applications are under more explicit user control and usually know the names of all needed files in advance.

7. EXPERIENCES

In the process of building and deploying GFS, we have experienced a variety of issues, some operational and some technical.

Initially, GFS was conceived as the backend file system for our production systems. Over time, the usage evolved to include research and development tasks. It started with little support for things like permissions and quotas but now includes rudimentary forms of these. While production systems are well disciplined and controlled, users sometimes are not. More infrastructure is required to keep users from interfering with one another.

Some of our biggest problems were disk and Linux related. Many of our disks claimed to the Linux driver that they supported a range of IDE protocol versions but in fact responded reliably only to the more recent ones. Since the protocol versions are very similar, these drives mostly worked, but occasionally the mismatches would cause the drive and the kernel to disagree about the drive's state. This would corrupt data silently due to problems in the kernel. This problem motivated our use of checksums to detect data corruption, while concurrently we modified the kernel to handle these protocol mismatches.

Earlier we had some problems with Linux 2.2 kernels due to the cost of `fsync()`. Its cost is proportional to the size of the file rather than the size of the modified portion. This was a problem for our large operation logs especially before we implemented checkpointing. We worked around this for a time by using synchronous writes and eventually migrated to Linux 2.4.

Another Linux problem was a single reader-writer lock which any thread in an address space must hold when it pages in from disk (reader lock) or modifies the address space in an `mmap()` call (writer lock). We saw transient timeouts in our system under light load and looked hard for resource bottlenecks or sporadic hardware failures. Eventually, we found that this single lock blocked the primary network thread from mapping new data into memory while the disk threads were paging in previously mapped data. Since we are mainly limited by the network interface rather than by memory copy bandwidth, we worked around this by replacing `mmap()` with `pread()` at the cost of an extra copy.

Despite occasional problems, the availability of Linux code has helped us time and again to explore and understand system behavior. When appropriate, we improve the kernel and share the changes with the open source community.

8. RELATED WORK

Like other large distributed file systems such as AFS [5], GFS provides a location independent namespace which enables data to be moved transparently for load balance or fault tolerance. Unlike AFS, GFS spreads a file's data across storage servers in a way more akin to xFS [1] and Swift [3] in order to deliver aggregate performance and increased fault tolerance.

As disks are relatively cheap and replication is simpler than more sophisticated RAID [9] approaches, GFS currently uses only replication for redundancy and so consumes more raw storage than xFS or Swift.

In contrast to systems like AFS, xFS, Frangipani [12], and Intermezzo [6], GFS does not provide any caching below the file system interface. Our target workloads have little reuse within a single application run because they either stream through a large data set or randomly seek within it and read small amounts of data each time.

Some distributed file systems like Frangipani, xFS, Minnesota's GFS[11] and GPFS [10] remove the centralized server and rely on distributed algorithms for consistency and management. We opt for the centralized approach in order to simplify the design, increase its reliability, and gain flexibility. In particular, a centralized master makes it much easier to implement sophisticated chunk placement and replication policies since the master already has most of the relevant information and controls how it changes. We address fault tolerance by keeping the master state small and fully replicated on other machines. Scalability and high availability (for reads) are currently provided by our shadow master mechanism. Updates to the master state are made persistent by appending to a write-ahead log. Therefore we could adapt a primary-copy scheme like the one in Harp [7] to provide high availability with stronger consistency guarantees than our current scheme.

We are addressing a problem similar to Lustre [8] in terms of delivering aggregate performance to a large number of clients. However, we have simplified the problem significantly by focusing on the needs of our applications rather than building a POSIX-compliant file system. Additionally, GFS assumes large number of unreliable components and so fault tolerance is central to our design.

GFS most closely resembles the NASD architecture [4]. While the NASD architecture is based on network-attached disk drives, GFS uses commodity machines as chunkservers, as done in the NASD prototype. Unlike the NASD work, our chunkservers use lazily allocated fixed-size chunks rather than variable-length objects. Additionally, GFS implements features such as rebalancing, replication, and recovery that are required in a production environment.

Unlike Minnesota's GFS and NASD, we do not seek to alter the model of the storage device. We focus on addressing day-to-day data processing needs for complicated distributed systems with existing commodity components.

The producer-consumer queues enabled by atomic record appends address a similar problem as the distributed queues in River [2]. While River uses memory-based queues distributed across machines and careful data flow control, GFS uses a persistent file that can be appended to concurrently by many producers. The River model supports m-to-n distributed queues but lacks the fault tolerance that comes with persistent storage, while GFS only supports m-to-1 queues efficiently. Multiple consumers can read the same file, but they must coordinate to partition the incoming load.

9. CONCLUSIONS

The Google File System demonstrates the qualities essential for supporting large-scale data processing workloads on commodity hardware. While some design decisions are specific to our unique setting, many may apply to data processing tasks of a similar magnitude and cost consciousness.

We started by reexamining traditional file system assumptions in light of our current and anticipated application workloads and technological environment. Our observations have led to radically different points in the design space. We treat component failures as the norm rather than the exception, optimize for huge files that are mostly appended to (perhaps concurrently) and then read (usually sequentially), and both extend and relax the standard file system interface to improve the overall system.

Our system provides fault tolerance by constant monitoring, replicating crucial data, and fast and automatic recovery. Chunk replication allows us to tolerate chunkserver

failures. The frequency of these failures motivated a novel online repair mechanism that regularly and transparently repairs the damage and compensates for lost replicas as soon as possible. Additionally, we use checksumming to detect data corruption at the disk or IDE subsystem level, which becomes all too common given the number of disks in the system.

Our design delivers high aggregate throughput to many concurrent readers and writers performing a variety of tasks. We achieve this by separating file system control, which passes through the master, from data transfer, which passes directly between chunkservers and clients. Master involvement in common operations is minimized by a large chunk size and by chunk leases, which delegates authority to primary replicas in data mutations. This makes possible a simple, centralized master that does not become a bottleneck. We believe that improvements in our networking stack will lift the current limitation on the write throughput seen by an individual client.

GFS has successfully met our storage needs and is widely used within Google as the storage platform for research and development as well as production data processing. It is an important tool that enables us to continue to innovate and attack problems on the scale of the entire web.

ACKNOWLEDGMENTS

We wish to thank the following people for their contributions to the system or the paper. Brain Bershad (our shepherd) and the anonymous reviewers gave us valuable comments and suggestions. Anurag Acharya, Jeff Dean, and David desJardins contributed to the early design. Fay Chang worked on comparison of replicas across chunkservers. Guy Edjlali worked on storage quota. Markus Gutschke worked on a testing framework and security enhancements. David Kramer worked on performance enhancements. Fay Chang, Urs Hoelzle, Max Ibel, Sharon Perl, Rob Pike, and Debby Wallach commented on earlier drafts of the paper. Many of our colleagues at Google bravely trusted their data to a new file system and gave us useful feedback. Yoshka helped with early testing.

REFERENCES

[1] Thomas Anderson, Michael Dahlin, Jeanna Neefe, David Patterson, Drew Roselli, and Randolph Wang. Serverless network file systems. In *Proceedings of the 15th ACM Symposium on Operating System Principles*, pages 109–126, Copper Mountain Resort, Colorado, December 1995.

[2] Remzi H. Arpaci-Dusseau, Eric Anderson, Noah Treuhaft, David E. Culler, Joseph M. Hellerstein, David Patterson, and Kathy Yelick. Cluster I/O with River: Making the fast case common. In *Proceedings of the Sixth Workshop on Input/Output in Parallel and Distributed Systems (IOPADS '99)*, pages 10–22, Atlanta, Georgia, May 1999.

[3] Luis-Felipe Cabrera and Darrell D. E. Long. Swift: Using distributed disk striping to provide high I/O data rates. *Computer Systems*, 4(4):405–436, 1991.

[4] Garth A. Gibson, David F. Nagle, Khalil Amiri, Jeff Butler, Fay W. Chang, Howard Gobioff, Charles Hardin, Erik Riedel, David Rochberg, and Jim Zelenka. A cost-effective, high-bandwidth storage architecture. In *Proceedings of the 8th Architectural Support for Programming Languages and Operating Systems*, pages 92–103, San Jose, California, October 1998.

[5] John Howard, Michael Kazar, Sherri Menees, David Nichols, Mahadev Satyanarayanan, Robert Sidebotham, and Michael West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.

[6] InterMezzo. http://www.inter-mezzo.org, 2003.

[7] Barbara Liskov, Sanjay Ghemawat, Robert Gruber, Paul Johnson, Liuba Shrira, and Michael Williams. Replication in the Harp file system. In *13th Symposium on Operating System Principles*, pages 226–238, Pacific Grove, CA, October 1991.

[8] Lustre. http://www.lustreorg, 2003.

[9] David A. Patterson, Garth A. Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, pages 109–116, Chicago, Illinois, September 1988.

[10] Frank Schmuck and Roger Haskin. GPFS: A shared-disk file system for large computing clusters. In *Proceedings of the First USENIX Conference on File and Storage Technologies*, pages 231–244, Monterey, California, January 2002.

[11] Steven R. Soltis, Thomas M. Ruwart, and Matthew T. O'Keefe. The Gobal File System. In *Proceedings of the Fifth NASA Goddard Space Flight Center Conference on Mass Storage Systems and Technologies*, College Park, Maryland, September 1996.

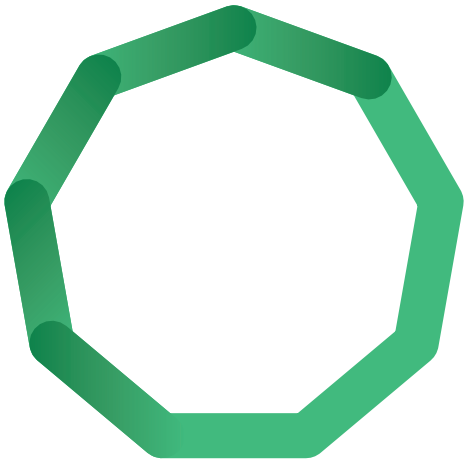
[12] Chandramohan A. Thekkath, Timothy Mann, and Edward K. Lee. Frangipani: A scalable distributed file system. In *Proceedings of the 16th ACM Symposium on Operating System Principles*, pages 224–237, Saint-Malo, France, October 1997.

Chapter 13

MapReduce: Simplified Data Processing on Large Clusters

Authors:
Jeffrey Dean, Sanjay Ghemawat

Publication year: 2004



MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

jeff@google.com, sanjay@google.com

Google, Inc.

Abstract

MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a *map* function that processes a key/value pair to generate a set of intermediate key/value pairs, and a *reduce* function that merges all intermediate values associated with the same intermediate key. Many real world tasks are expressible in this model, as shown in the paper.

Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. The run-time system takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures, and managing the required inter-machine communication. This allows programmers without any experience with parallel and distributed systems to easily utilize the resources of a large distributed system.

Our implementation of MapReduce runs on a large cluster of commodity machines and is highly scalable: a typical MapReduce computation processes many terabytes of data on thousands of machines. Programmers find the system easy to use: hundreds of MapReduce programs have been implemented and upwards of one thousand MapReduce jobs are executed on Google's clusters every day.

1 Introduction

Over the past five years, the authors and many others at Google have implemented hundreds of special-purpose computations that process large amounts of raw data, such as crawled documents, web request logs, etc., to compute various kinds of derived data, such as inverted indices, various representations of the graph structure of web documents, summaries of the number of pages crawled per host, the set of most frequent queries in a given day, etc. Most such computations are conceptually straightforward. However, the input data is usually large and the computations have to be distributed across hundreds or thousands of machines in order to finish in a reasonable amount of time. The issues of how to parallelize the computation, distribute the data, and handle failures conspire to obscure the original simple computation with large amounts of complex code to deal with these issues.

As a reaction to this complexity, we designed a new abstraction that allows us to express the simple computations we were trying to perform but hides the messy details of parallelization, fault-tolerance, data distribution and load balancing in a library. Our abstraction is inspired by the *map* and *reduce* primitives present in Lisp and many other functional languages. We realized that most of our computations involved applying a *map* operation to each logical "record" in our input in order to compute a set of intermediate key/value pairs, and then applying a *reduce* operation to all the values that shared the same key, in order to combine the derived data appropriately. Our use of a functional model with user-specified map and reduce operations allows us to parallelize large computations easily and to use re-execution as the primary mechanism for fault tolerance.

The major contributions of this work are a simple and powerful interface that enables automatic parallelization and distribution of large-scale computations, combined with an implementation of this interface that achieves high performance on large clusters of commodity PCs.

Section 2 describes the basic programming model and gives several examples. Section 3 describes an implementation of the MapReduce interface tailored towards our cluster-based computing environment. Section 4 describes several refinements of the programming model that we have found useful. Section 5 has performance measurements of our implementation for a variety of tasks. Section 6 explores the use of MapReduce within Google including our experiences in using it as the basis

To appear in OSDI 2004

for a rewrite of our production indexing system. Section 7 discusses related and future work.

2 Programming Model

The computation takes a set of *input* key/value pairs, and produces a set of *output* key/value pairs. The user of the MapReduce library expresses the computation as two functions: *Map* and *Reduce*.

Map, written by the user, takes an input pair and produces a set of *intermediate* key/value pairs. The MapReduce library groups together all intermediate values associated with the same intermediate key I and passes them to the *Reduce* function.

The *Reduce* function, also written by the user, accepts an intermediate key I and a set of values for that key. It merges together these values to form a possibly smaller set of values. Typically just zero or one output value is produced per *Reduce* invocation. The intermediate values are supplied to the user's reduce function via an iterator. This allows us to handle lists of values that are too large to fit in memory.

2.1 Example

Consider the problem of counting the number of occurrences of each word in a large collection of documents. The user would write code similar to the following pseudo-code:

```
map(String key, String value):
  // key: document name
  // value: document contents
  for each word w in value:
    EmitIntermediate(w, "1");

reduce(String key, Iterator values):
  // key: a word
  // values: a list of counts
  int result = 0;
  for each v in values:
    result += ParseInt(v);
  Emit(AsString(result));
```

The `map` function emits each word plus an associated count of occurrences (just '1' in this simple example). The `reduce` function sums together all counts emitted for a particular word.

In addition, the user writes code to fill in a *mapreduce specification* object with the names of the input and output files, and optional tuning parameters. The user then invokes the *MapReduce* function, passing it the specification object. The user's code is linked together with the MapReduce library (implemented in C++). Appendix A contains the full program text for this example.

2.2 Types

Even though the previous pseudo-code is written in terms of string inputs and outputs, conceptually the map and reduce functions supplied by the user have associated types:

map	`(k1,v1)`	$\rightarrow$ `list(k2,v2)`
reduce	`(k2,list(v2))`	$\rightarrow$ `list(v2)`

I.e., the input keys and values are drawn from a different domain than the output keys and values. Furthermore, the intermediate keys and values are from the same domain as the output keys and values.

Our C++ implementation passes strings to and from the user-defined functions and leaves it to the user code to convert between strings and appropriate types.

2.3 More Examples

Here are a few simple examples of interesting programs that can be easily expressed as MapReduce computations.

Distributed Grep: The map function emits a line if it matches a supplied pattern. The reduce function is an identity function that just copies the supplied intermediate data to the output.

Count of URL Access Frequency: The map function processes logs of web page requests and outputs $\langle$`URL`, `1`$\rangle$. The reduce function adds together all values for the same URL and emits a $\langle$`URL`, `total count`$\rangle$ pair.

Reverse Web-Link Graph: The map function outputs $\langle$`target`, `source`$\rangle$ pairs for each link to a `target` URL found in a page named `source`. The reduce function concatenates the list of all source URLs associated with a given target URL and emits the pair: $\langle$`target`, $list$(`source`)$\rangle$

Term-Vector per Host: A term vector summarizes the most important words that occur in a document or a set of documents as a list of $\langle word, frequency \rangle$ pairs. The map function emits a $\langle$`hostname`, `term vector`$\rangle$ pair for each input document (where the hostname is extracted from the URL of the document). The reduce function is passed all per-document term vectors for a given host. It adds these term vectors together, throwing away infrequent terms, and then emits a final $\langle$`hostname`, `term vector`$\rangle$ pair.

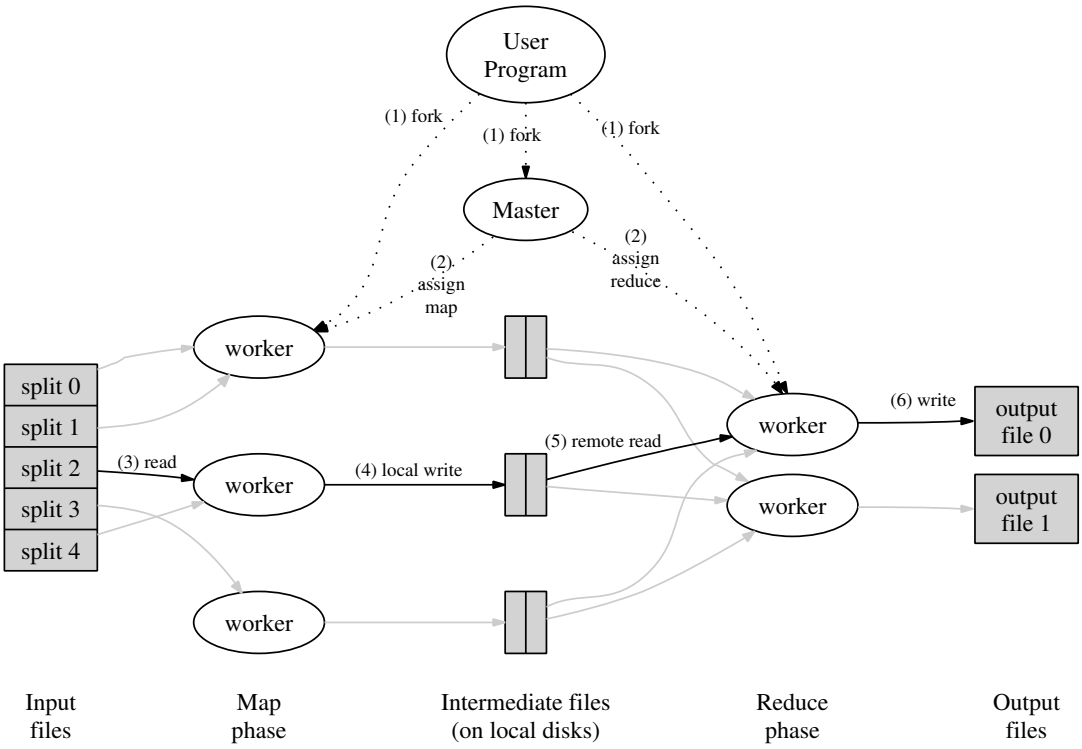


Figure 1: Execution overview

Inverted Index: The map function parses each document, and emits a sequence of $\langle \texttt{word}, \texttt{document ID} \rangle$ pairs. The reduce function accepts all pairs for a given word, sorts the corresponding document IDs and emits a $\langle \texttt{word}, list(\texttt{document ID}) \rangle$ pair. The set of all output pairs forms a simple inverted index. It is easy to augment this computation to keep track of word positions.

Distributed Sort: The map function extracts the key from each record, and emits a $\langle \texttt{key}, \texttt{record} \rangle$ pair. The reduce function emits all pairs unchanged. This computation depends on the partitioning facilities described in Section 4.1 and the ordering properties described in Section 4.2.

3 Implementation

Many different implementations of the MapReduce interface are possible. The right choice depends on the environment. For example, one implementation may be suitable for a small shared-memory machine, another for a large NUMA multi-processor, and yet another for an even larger collection of networked machines.

This section describes an implementation targeted to the computing environment in wide use at Google: large clusters of commodity PCs connected together with switched Ethernet [4]. In our environment:

(1) Machines are typically dual-processor x86 processors running Linux, with 2-4 GB of memory per machine.

(2) Commodity networking hardware is used – typically either 100 megabits/second or 1 gigabit/second at the machine level, but averaging considerably less in overall bisection bandwidth.

(3) A cluster consists of hundreds or thousands of machines, and therefore machine failures are common.

(4) Storage is provided by inexpensive IDE disks attached directly to individual machines. A distributed file system [8] developed in-house is used to manage the data stored on these disks. The file system uses replication to provide availability and reliability on top of unreliable hardware.

(5) Users submit jobs to a scheduling system. Each job consists of a set of tasks, and is mapped by the scheduler to a set of available machines within a cluster.

3.1 Execution Overview

The *Map* invocations are distributed across multiple machines by automatically partitioning the input data

into a set of M *splits*. The input splits can be processed in parallel by different machines. *Reduce* invocations are distributed by partitioning the intermediate key space into R pieces using a partitioning function (e.g., $hash(key) \mathbf{mod} R$). The number of partitions (R) and the partitioning function are specified by the user.

Figure 1 shows the overall flow of a MapReduce operation in our implementation. When the user program calls the `MapReduce` function, the following sequence of actions occurs (the numbered labels in Figure 1 correspond to the numbers in the list below):

1. The MapReduce library in the user program first splits the input files into M pieces of typically 16 megabytes to 64 megabytes (MB) per piece (controllable by the user via an optional parameter). It then starts up many copies of the program on a cluster of machines.

2. One of the copies of the program is special – the master. The rest are workers that are assigned work by the master. There are M map tasks and R reduce tasks to assign. The master picks idle workers and assigns each one a map task or a reduce task.

3. A worker who is assigned a map task reads the contents of the corresponding input split. It parses key/value pairs out of the input data and passes each pair to the user-defined *Map* function. The intermediate key/value pairs produced by the *Map* function are buffered in memory.

4. Periodically, the buffered pairs are written to local disk, partitioned into R regions by the partitioning function. The locations of these buffered pairs on the local disk are passed back to the master, who is responsible for forwarding these locations to the reduce workers.

5. When a reduce worker is notified by the master about these locations, it uses remote procedure calls to read the buffered data from the local disks of the map workers. When a reduce worker has read all intermediate data, it sorts it by the intermediate keys so that all occurrences of the same key are grouped together. The sorting is needed because typically many different keys map to the same reduce task. If the amount of intermediate data is too large to fit in memory, an external sort is used.

6. The reduce worker iterates over the sorted intermediate data and for each unique intermediate key encountered, it passes the key and the corresponding set of intermediate values to the user's *Reduce* function. The output of the *Reduce* function is appended to a final output file for this reduce partition.

7. When all map tasks and reduce tasks have been completed, the master wakes up the user program. At this point, the `MapReduce` call in the user program returns back to the user code.

After successful completion, the output of the mapreduce execution is available in the R output files (one per reduce task, with file names as specified by the user). Typically, users do not need to combine these R output files into one file – they often pass these files as input to another MapReduce call, or use them from another distributed application that is able to deal with input that is partitioned into multiple files.

3.2 Master Data Structures

The master keeps several data structures. For each map task and reduce task, it stores the state (*idle*, *in-progress*, or *completed*), and the identity of the worker machine (for non-idle tasks).

The master is the conduit through which the location of intermediate file regions is propagated from map tasks to reduce tasks. Therefore, for each completed map task, the master stores the locations and sizes of the R intermediate file regions produced by the map task. Updates to this location and size information are received as map tasks are completed. The information is pushed incrementally to workers that have *in-progress* reduce tasks.

3.3 Fault Tolerance

Since the MapReduce library is designed to help process very large amounts of data using hundreds or thousands of machines, the library must tolerate machine failures gracefully.

Worker Failure

The master pings every worker periodically. If no response is received from a worker in a certain amount of time, the master marks the worker as failed. Any map tasks completed by the worker are reset back to their initial *idle* state, and therefore become eligible for scheduling on other workers. Similarly, any map task or reduce task in progress on a failed worker is also reset to *idle* and becomes eligible for rescheduling.

Completed map tasks are re-executed on a failure because their output is stored on the local disk(s) of the failed machine and is therefore inaccessible. Completed reduce tasks do not need to be re-executed since their output is stored in a global file system.

When a map task is executed first by worker A and then later executed by worker B (because A failed), all

workers executing reduce tasks are notified of the re-execution. Any reduce task that has not already read the data from worker A will read the data from worker B.

MapReduce is resilient to large-scale worker failures. For example, during one MapReduce operation, network maintenance on a running cluster was causing groups of 80 machines at a time to become unreachable for several minutes. The MapReduce master simply re-executed the work done by the unreachable worker machines, and continued to make forward progress, eventually completing the MapReduce operation.

Master Failure

It is easy to make the master write periodic checkpoints of the master data structures described above. If the master task dies, a new copy can be started from the last checkpointed state. However, given that there is only a single master, its failure is unlikely; therefore our current implementation aborts the MapReduce computation if the master fails. Clients can check for this condition and retry the MapReduce operation if they desire.

Semantics in the Presence of Failures

When the user-supplied *map* and *reduce* operators are deterministic functions of their input values, our distributed implementation produces the same output as would have been produced by a non-faulting sequential execution of the entire program.

We rely on atomic commits of map and reduce task outputs to achieve this property. Each in-progress task writes its output to private temporary files. A reduce task produces one such file, and a map task produces R such files (one per reduce task). When a map task completes, the worker sends a message to the master and includes the names of the R temporary files in the message. If the master receives a completion message for an already completed map task, it ignores the message. Otherwise, it records the names of R files in a master data structure.

When a reduce task completes, the reduce worker atomically renames its temporary output file to the final output file. If the same reduce task is executed on multiple machines, multiple rename calls will be executed for the same final output file. We rely on the atomic rename operation provided by the underlying file system to guarantee that the final file system state contains just the data produced by one execution of the reduce task.

The vast majority of our *map* and *reduce* operators are deterministic, and the fact that our semantics are equivalent to a sequential execution in this case makes it very easy for programmers to reason about their program's behavior. When the *map* and/or *reduce* operators are non-deterministic, we provide weaker but still reasonable semantics. In the presence of non-deterministic operators, the output of a particular reduce task R_1 is equivalent to the output for R_1 produced by a sequential execution of the non-deterministic program. However, the output for a different reduce task R_2 may correspond to the output for R_2 produced by a different sequential execution of the non-deterministic program.

Consider map task M and reduce tasks R_1 and R_2. Let $e(R_i)$ be the execution of R_i that committed (there is exactly one such execution). The weaker semantics arise because $e(R_1)$ may have read the output produced by one execution of M and $e(R_2)$ may have read the output produced by a different execution of M.

3.4 Locality

Network bandwidth is a relatively scarce resource in our computing environment. We conserve network bandwidth by taking advantage of the fact that the input data (managed by GFS [8]) is stored on the local disks of the machines that make up our cluster. GFS divides each file into 64 MB blocks, and stores several copies of each block (typically 3 copies) on different machines. The MapReduce master takes the location information of the input files into account and attempts to schedule a map task on a machine that contains a replica of the corresponding input data. Failing that, it attempts to schedule a map task near a replica of that task's input data (e.g., on a worker machine that is on the same network switch as the machine containing the data). When running large MapReduce operations on a significant fraction of the workers in a cluster, most input data is read locally and consumes no network bandwidth.

3.5 Task Granularity

We subdivide the map phase into M pieces and the reduce phase into R pieces, as described above. Ideally, M and R should be much larger than the number of worker machines. Having each worker perform many different tasks improves dynamic load balancing, and also speeds up recovery when a worker fails: the many map tasks it has completed can be spread out across all the other worker machines.

There are practical bounds on how large M and R can be in our implementation, since the master must make $O(M + R)$ scheduling decisions and keeps $O(M * R)$ state in memory as described above. (The constant factors for memory usage are small however: the $O(M * R)$ piece of the state consists of approximately one byte of data per map task/reduce task pair.)

Furthermore, R is often constrained by users because the output of each reduce task ends up in a separate output file. In practice, we tend to choose M so that each individual task is roughly 16 MB to 64 MB of input data (so that the locality optimization described above is most effective), and we make R a small multiple of the number of worker machines we expect to use. We often perform MapReduce computations with $M = 200,000$ and $R = 5,000$, using 2,000 worker machines.

3.6 Backup Tasks

One of the common causes that lengthens the total time taken for a MapReduce operation is a "straggler": a machine that takes an unusually long time to complete one of the last few map or reduce tasks in the computation. Stragglers can arise for a whole host of reasons. For example, a machine with a bad disk may experience frequent correctable errors that slow its read performance from 30 MB/s to 1 MB/s. The cluster scheduling system may have scheduled other tasks on the machine, causing it to execute the MapReduce code more slowly due to competition for CPU, memory, local disk, or network bandwidth. A recent problem we experienced was a bug in machine initialization code that caused processor caches to be disabled: computations on affected machines slowed down by over a factor of one hundred.

We have a general mechanism to alleviate the problem of stragglers. When a MapReduce operation is close to completion, the master schedules backup executions of the remaining *in-progress* tasks. The task is marked as completed whenever either the primary or the backup execution completes. We have tuned this mechanism so that it typically increases the computational resources used by the operation by no more than a few percent. We have found that this significantly reduces the time to complete large MapReduce operations. As an example, the sort program described in Section 5.3 takes 44% longer to complete when the backup task mechanism is disabled.

4 Refinements

Although the basic functionality provided by simply writing *Map* and *Reduce* functions is sufficient for most needs, we have found a few extensions useful. These are described in this section.

4.1 Partitioning Function

The users of MapReduce specify the number of reduce tasks/output files that they desire (R). Data gets partitioned across these tasks using a partitioning function on the intermediate key. A default partitioning function is provided that uses hashing (e.g. "$hash(key) \bmod R$"). This tends to result in fairly well-balanced partitions. In some cases, however, it is useful to partition data by some other function of the key. For example, sometimes the output keys are URLs, and we want all entries for a single host to end up in the same output file. To support situations like this, the user of the MapReduce library can provide a special partitioning function. For example, using "$hash(Hostname(urlkey)) \bmod R$" as the partitioning function causes all URLs from the same host to end up in the same output file.

4.2 Ordering Guarantees

We guarantee that within a given partition, the intermediate key/value pairs are processed in increasing key order. This ordering guarantee makes it easy to generate a sorted output file per partition, which is useful when the output file format needs to support efficient random access lookups by key, or users of the output find it convenient to have the data sorted.

4.3 Combiner Function

In some cases, there is significant repetition in the intermediate keys produced by each map task, and the user-specified *Reduce* function is commutative and associative. A good example of this is the word counting example in Section 2.1. Since word frequencies tend to follow a Zipf distribution, each map task will produce hundreds or thousands of records of the form `<the, 1>`. All of these counts will be sent over the network to a single reduce task and then added together by the *Reduce* function to produce one number. We allow the user to specify an optional *Combiner* function that does partial merging of this data before it is sent over the network.

The *Combiner* function is executed on each machine that performs a map task. Typically the same code is used to implement both the combiner and the reduce functions. The only difference between a reduce function and a combiner function is how the MapReduce library handles the output of the function. The output of a reduce function is written to the final output file. The output of a combiner function is written to an intermediate file that will be sent to a reduce task.

Partial combining significantly speeds up certain classes of MapReduce operations. Appendix A contains an example that uses a combiner.

4.4 Input and Output Types

The MapReduce library provides support for reading input data in several different formats. For example, "text"

mode input treats each line as a key/value pair: the key is the offset in the file and the value is the contents of the line. Another common supported format stores a sequence of key/value pairs sorted by key. Each input type implementation knows how to split itself into meaningful ranges for processing as separate map tasks (e.g. text mode's range splitting ensures that range splits occur only at line boundaries). Users can add support for a new input type by providing an implementation of a simple *reader* interface, though most users just use one of a small number of predefined input types.

A *reader* does not necessarily need to provide data read from a file. For example, it is easy to define a *reader* that reads records from a database, or from data structures mapped in memory.

In a similar fashion, we support a set of output types for producing data in different formats and it is easy for user code to add support for new output types.

4.5 Side-effects

In some cases, users of MapReduce have found it convenient to produce auxiliary files as additional outputs from their map and/or reduce operators. We rely on the application writer to make such side-effects atomic and idempotent. Typically the application writes to a temporary file and atomically renames this file once it has been fully generated.

We do not provide support for atomic two-phase commits of multiple output files produced by a single task. Therefore, tasks that produce multiple output files with cross-file consistency requirements should be deterministic. This restriction has never been an issue in practice.

4.6 Skipping Bad Records

Sometimes there are bugs in user code that cause the *Map* or *Reduce* functions to crash deterministically on certain records. Such bugs prevent a MapReduce operation from completing. The usual course of action is to fix the bug, but sometimes this is not feasible; perhaps the bug is in a third-party library for which source code is unavailable. Also, sometimes it is acceptable to ignore a few records, for example when doing statistical analysis on a large data set. We provide an optional mode of execution where the MapReduce library detects which records cause deterministic crashes and skips these records in order to make forward progress.

Each worker process installs a signal handler that catches segmentation violations and bus errors. Before invoking a user *Map* or *Reduce* operation, the MapReduce library stores the sequence number of the argument in a global variable. If the user code generates a signal, the signal handler sends a "last gasp" UDP packet that contains the sequence number to the MapReduce master. When the master has seen more than one failure on a particular record, it indicates that the record should be skipped when it issues the next re-execution of the corresponding Map or Reduce task.

4.7 Local Execution

Debugging problems in *Map* or *Reduce* functions can be tricky, since the actual computation happens in a distributed system, often on several thousand machines, with work assignment decisions made dynamically by the master. To help facilitate debugging, profiling, and small-scale testing, we have developed an alternative implementation of the MapReduce library that sequentially executes all of the work for a MapReduce operation on the local machine. Controls are provided to the user so that the computation can be limited to particular map tasks. Users invoke their program with a special flag and can then easily use any debugging or testing tools they find useful (e.g. `gdb`).

4.8 Status Information

The master runs an internal HTTP server and exports a set of status pages for human consumption. The status pages show the progress of the computation, such as how many tasks have been completed, how many are in progress, bytes of input, bytes of intermediate data, bytes of output, processing rates, etc. The pages also contain links to the standard error and standard output files generated by each task. The user can use this data to predict how long the computation will take, and whether or not more resources should be added to the computation. These pages can also be used to figure out when the computation is much slower than expected.

In addition, the top-level status page shows which workers have failed, and which map and reduce tasks they were processing when they failed. This information is useful when attempting to diagnose bugs in the user code.

4.9 Counters

The MapReduce library provides a counter facility to count occurrences of various events. For example, user code may want to count total number of words processed or the number of German documents indexed, etc.

To use this facility, user code creates a named counter object and then increments the counter appropriately in the *Map* and/or *Reduce* function. For example:

```
Counter* uppercase;
uppercase = GetCounter("uppercase");

map(String name, String contents):
  for each word w in contents:
    if (IsCapitalized(w)):
      uppercase->Increment();
    EmitIntermediate(w, "1");
```

The counter values from individual worker machines are periodically propagated to the master (piggybacked on the ping response). The master aggregates the counter values from successful map and reduce tasks and returns them to the user code when the MapReduce operation is completed. The current counter values are also displayed on the master status page so that a human can watch the progress of the live computation. When aggregating counter values, the master eliminates the effects of duplicate executions of the same map or reduce task to avoid double counting. (Duplicate executions can arise from our use of backup tasks and from re-execution of tasks due to failures.)

Some counter values are automatically maintained by the MapReduce library, such as the number of input key/value pairs processed and the number of output key/value pairs produced.

Users have found the counter facility useful for sanity checking the behavior of MapReduce operations. For example, in some MapReduce operations, the user code may want to ensure that the number of output pairs produced exactly equals the number of input pairs processed, or that the fraction of German documents processed is within some tolerable fraction of the total number of documents processed.

5 Performance

In this section we measure the performance of MapReduce on two computations running on a large cluster of machines. One computation searches through approximately one terabyte of data looking for a particular pattern. The other computation sorts approximately one terabyte of data.

These two programs are representative of a large subset of the real programs written by users of MapReduce – one class of programs shuffles data from one representation to another, and another class extracts a small amount of interesting data from a large data set.

5.1 Cluster Configuration

All of the programs were executed on a cluster that consisted of approximately 1800 machines. Each machine had two 2GHz Intel Xeon processors with Hyper-Threading enabled, 4GB of memory, two 160GB IDE disks, and a gigabit Ethernet link. The machines were arranged in a two-level tree-shaped switched network with approximately 100-200 Gbps of aggregate bandwidth available at the root. All of the machines were in the same hosting facility and therefore the round-trip time between any pair of machines was less than a millisecond.

Out of the 4GB of memory, approximately 1-1.5GB was reserved by other tasks running on the cluster. The programs were executed on a weekend afternoon, when the CPUs, disks, and network were mostly idle.

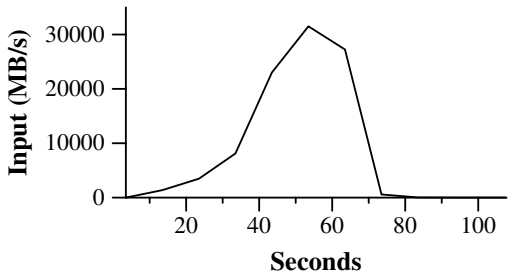


Figure 2: Data transfer rate over time

5.2 Grep

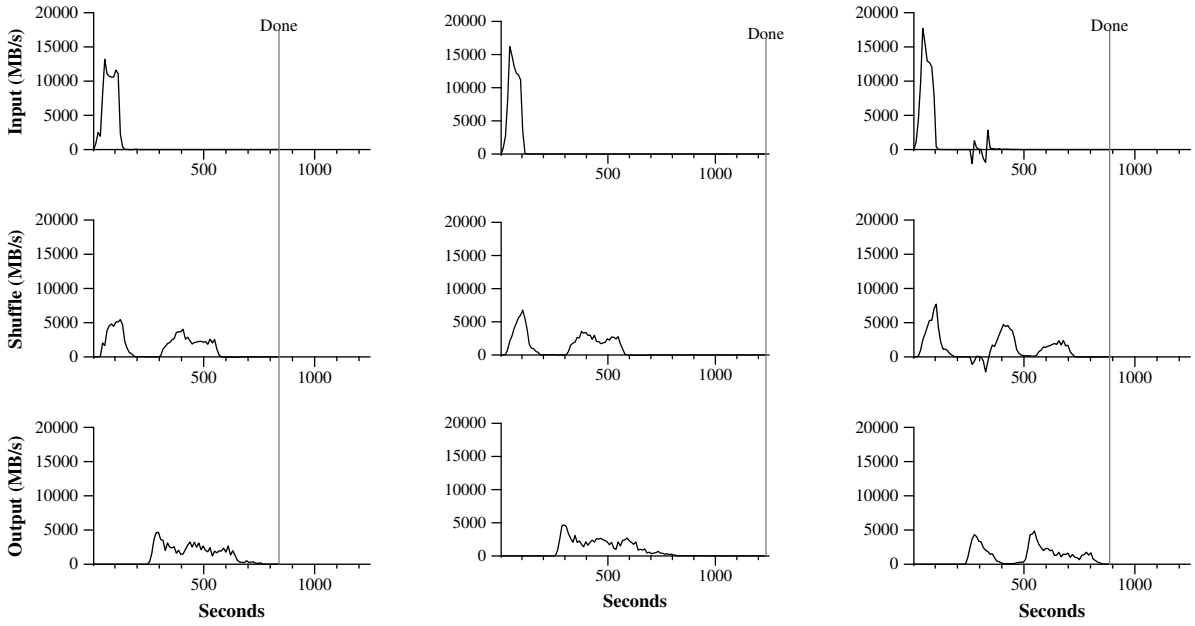
The *grep* program scans through 10^{10} 100-byte records, searching for a relatively rare three-character pattern (the pattern occurs in 92,337 records). The input is split into approximately 64MB pieces ($M = 15000$), and the entire output is placed in one file ($R = 1$).

Figure 2 shows the progress of the computation over time. The Y-axis shows the rate at which the input data is scanned. The rate gradually picks up as more machines are assigned to this MapReduce computation, and peaks at over 30 GB/s when 1764 workers have been assigned. As the map tasks finish, the rate starts dropping and hits zero about 80 seconds into the computation. The entire computation takes approximately 150 seconds from start to finish. This includes about a minute of startup overhead. The overhead is due to the propagation of the program to all worker machines, and delays interacting with GFS to open the set of 1000 input files and to get the information needed for the locality optimization.

5.3 Sort

The *sort* program sorts 10^{10} 100-byte records (approximately 1 terabyte of data). This program is modeled after the TeraSort benchmark [10].

The sorting program consists of less than 50 lines of user code. A three-line *Map* function extracts a 10-byte sorting key from a text line and emits the key and the



(a) Normal execution (b) No backup tasks (c) 200 tasks killed

Figure 3: Data transfer rates over time for different executions of the sort program

original text line as the intermediate key/value pair. We used a built-in *Identity* function as the *Reduce* operator. This functions passes the intermediate key/value pair unchanged as the output key/value pair. The final sorted output is written to a set of 2-way replicated GFS files (i.e., 2 terabytes are written as the output of the program).

As before, the input data is split into 64MB pieces ($M = 15000$). We partition the sorted output into 4000 files ($R = 4000$). The partitioning function uses the initial bytes of the key to segregate it into one of R pieces.

Our partitioning function for this benchmark has built-in knowledge of the distribution of keys. In a general sorting program, we would add a pre-pass MapReduce operation that would collect a sample of the keys and use the distribution of the sampled keys to compute split-points for the final sorting pass.

Figure 3 (a) shows the progress of a normal execution of the sort program. The top-left graph shows the rate at which input is read. The rate peaks at about 13 GB/s and dies off fairly quickly since all map tasks finish before 200 seconds have elapsed. Note that the input rate is less than for *grep*. This is because the sort map tasks spend about half their time and I/O bandwidth writing intermediate output to their local disks. The corresponding intermediate output for grep had negligible size.

The middle-left graph shows the rate at which data is sent over the network from the map tasks to the reduce tasks. This shuffling starts as soon as the first map task completes. The first hump in the graph is for the first batch of approximately 1700 reduce tasks (the entire MapReduce was assigned about 1700 machines, and each machine executes at most one reduce task at a time). Roughly 300 seconds into the computation, some of these first batch of reduce tasks finish and we start shuffling data for the remaining reduce tasks. All of the shuffling is done about 600 seconds into the computation.

The bottom-left graph shows the rate at which sorted data is written to the final output files by the reduce tasks. There is a delay between the end of the first shuffling period and the start of the writing period because the machines are busy sorting the intermediate data. The writes continue at a rate of about 2-4 GB/s for a while. All of the writes finish about 850 seconds into the computation. Including startup overhead, the entire computation takes 891 seconds. This is similar to the current best reported result of 1057 seconds for the TeraSort benchmark [18].

A few things to note: the input rate is higher than the shuffle rate and the output rate because of our locality optimization – most data is read from a local disk and bypasses our relatively bandwidth constrained network. The shuffle rate is higher than the output rate because the output phase writes two copies of the sorted data (we make two replicas of the output for reliability and availability reasons). We write two replicas because that is the mechanism for reliability and availability provided by our underlying file system. Network bandwidth requirements for writing data would be reduced if the underlying file system used erasure coding [14] rather than replication.

5.4 Effect of Backup Tasks

In Figure 3 (b), we show an execution of the sort program with backup tasks disabled. The execution flow is similar to that shown in Figure 3 (a), except that there is a very long tail where hardly any write activity occurs. After 960 seconds, all except 5 of the reduce tasks are completed. However these last few stragglers don't finish until 300 seconds later. The entire computation takes 1283 seconds, an increase of 44% in elapsed time.

5.5 Machine Failures

In Figure 3 (c), we show an execution of the sort program where we intentionally killed 200 out of 1746 worker processes several minutes into the computation. The underlying cluster scheduler immediately restarted new worker processes on these machines (since only the processes were killed, the machines were still functioning properly).

The worker deaths show up as a negative input rate since some previously completed map work disappears (since the corresponding map workers were killed) and needs to be redone. The re-execution of this map work happens relatively quickly. The entire computation finishes in 933 seconds including startup overhead (just an increase of 5% over the normal execution time).

6 Experience

We wrote the first version of the MapReduce library in February of 2003, and made significant enhancements to it in August of 2003, including the locality optimization, dynamic load balancing of task execution across worker machines, etc. Since that time, we have been pleasantly surprised at how broadly applicable the MapReduce library has been for the kinds of problems we work on. It has been used across a wide range of domains within Google, including:

- large-scale machine learning problems,
- clustering problems for the Google News and Froogle products,
- extraction of data used to produce reports of popular queries (e.g. Google Zeitgeist),
- extraction of properties of web pages for new experiments and products (e.g. extraction of geographical locations from a large corpus of web pages for localized search), and
- large-scale graph computations.

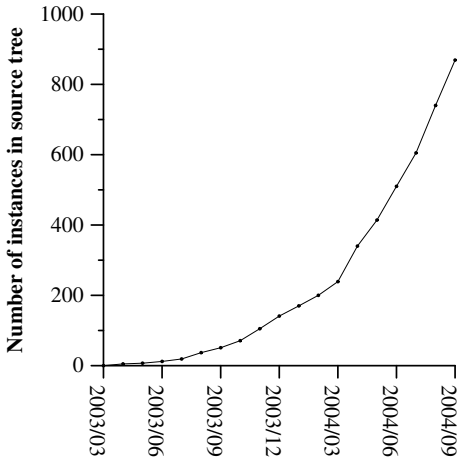


Figure 4: MapReduce instances over time

Number of jobs	29,423
Average job completion time	634 secs
Machine days used	79,186 days
Input data read	3,288 TB
Intermediate data produced	758 TB
Output data written	193 TB
Average worker machines per job	157
Average worker deaths per job	1.2
Average map tasks per job	3,351
Average reduce tasks per job	55
Unique *map* implementations	395
Unique *reduce* implementations	269
Unique *map/reduce* combinations	426

Table 1: MapReduce jobs run in August 2004

Figure 4 shows the significant growth in the number of separate MapReduce programs checked into our primary source code management system over time, from 0 in early 2003 to almost 900 separate instances as of late September 2004. MapReduce has been so successful because it makes it possible to write a simple program and run it efficiently on a thousand machines in the course of half an hour, greatly speeding up the development and prototyping cycle. Furthermore, it allows programmers who have no experience with distributed and/or parallel systems to exploit large amounts of resources easily.

At the end of each job, the MapReduce library logs statistics about the computational resources used by the job. In Table 1, we show some statistics for a subset of MapReduce jobs run at Google in August 2004.

6.1 Large-Scale Indexing

One of our most significant uses of MapReduce to date has been a complete rewrite of the production index-

ing system that produces the data structures used for the Google web search service. The indexing system takes as input a large set of documents that have been retrieved by our crawling system, stored as a set of GFS files. The raw contents for these documents are more than 20 terabytes of data. The indexing process runs as a sequence of five to ten MapReduce operations. Using MapReduce (instead of the ad-hoc distributed passes in the prior version of the indexing system) has provided several benefits:

- The indexing code is simpler, smaller, and easier to understand, because the code that deals with fault tolerance, distribution and parallelization is hidden within the MapReduce library. For example, the size of one phase of the computation dropped from approximately 3800 lines of C++ code to approximately 700 lines when expressed using MapReduce.
- The performance of the MapReduce library is good enough that we can keep conceptually unrelated computations separate, instead of mixing them together to avoid extra passes over the data. This makes it easy to change the indexing process. For example, one change that took a few months to make in our old indexing system took only a few days to implement in the new system.
- The indexing process has become much easier to operate, because most of the problems caused by machine failures, slow machines, and networking hiccups are dealt with automatically by the MapReduce library without operator intervention. Furthermore, it is easy to improve the performance of the indexing process by adding new machines to the indexing cluster.

7 Related Work

Many systems have provided restricted programming models and used the restrictions to parallelize the computation automatically. For example, an associative function can be computed over all prefixes of an N element array in $\log N$ time on N processors using parallel prefix computations [6, 9, 13]. MapReduce can be considered a simplification and distillation of some of these models based on our experience with large real-world computations. More significantly, we provide a fault-tolerant implementation that scales to thousands of processors. In contrast, most of the parallel processing systems have only been implemented on smaller scales and leave the details of handling machine failures to the programmer.

Bulk Synchronous Programming [17] and some MPI primitives [11] provide higher-level abstractions that make it easier for programmers to write parallel programs. A key difference between these systems and MapReduce is that MapReduce exploits a restricted programming model to parallelize the user program automatically and to provide transparent fault-tolerance.

Our locality optimization draws its inspiration from techniques such as active disks [12, 15], where computation is pushed into processing elements that are close to local disks, to reduce the amount of data sent across I/O subsystems or the network. We run on commodity processors to which a small number of disks are directly connected instead of running directly on disk controller processors, but the general approach is similar.

Our backup task mechanism is similar to the eager scheduling mechanism employed in the Charlotte System [3]. One of the shortcomings of simple eager scheduling is that if a given task causes repeated failures, the entire computation fails to complete. We fix some instances of this problem with our mechanism for skipping bad records.

The MapReduce implementation relies on an in-house cluster management system that is responsible for distributing and running user tasks on a large collection of shared machines. Though not the focus of this paper, the cluster management system is similar in spirit to other systems such as Condor [16].

The sorting facility that is a part of the MapReduce library is similar in operation to NOW-Sort [1]. Source machines (map workers) partition the data to be sorted and send it to one of R reduce workers. Each reduce worker sorts its data locally (in memory if possible). Of course NOW-Sort does not have the user-definable Map and Reduce functions that make our library widely applicable.

River [2] provides a programming model where processes communicate with each other by sending data over distributed queues. Like MapReduce, the River system tries to provide good average case performance even in the presence of non-uniformities introduced by heterogeneous hardware or system perturbations. River achieves this by careful scheduling of disk and network transfers to achieve balanced completion times. MapReduce has a different approach. By restricting the programming model, the MapReduce framework is able to partition the problem into a large number of fine-grained tasks. These tasks are dynamically scheduled on available workers so that faster workers process more tasks. The restricted programming model also allows us to schedule redundant executions of tasks near the end of the job which greatly reduces completion time in the presence of non-uniformities (such as slow or stuck workers).

BAD-FS [5] has a very different programming model from MapReduce, and unlike MapReduce, is targeted to

the execution of jobs across a wide-area network. However, there are two fundamental similarities. (1) Both systems use redundant execution to recover from data loss caused by failures. (2) Both use locality-aware scheduling to reduce the amount of data sent across congested network links.

TACC [7] is a system designed to simplify construction of highly-available networked services. Like MapReduce, it relies on re-execution as a mechanism for implementing fault-tolerance.

8 Conclusions

The MapReduce programming model has been successfully used at Google for many different purposes. We attribute this success to several reasons. First, the model is easy to use, even for programmers without experience with parallel and distributed systems, since it hides the details of parallelization, fault-tolerance, locality optimization, and load balancing. Second, a large variety of problems are easily expressible as MapReduce computations. For example, MapReduce is used for the generation of data for Google's production web search service, for sorting, for data mining, for machine learning, and many other systems. Third, we have developed an implementation of MapReduce that scales to large clusters of machines comprising thousands of machines. The implementation makes efficient use of these machine resources and therefore is suitable for use on many of the large computational problems encountered at Google.

We have learned several things from this work. First, restricting the programming model makes it easy to parallelize and distribute computations and to make such computations fault-tolerant. Second, network bandwidth is a scarce resource. A number of optimizations in our system are therefore targeted at reducing the amount of data sent across the network: the locality optimization allows us to read data from local disks, and writing a single copy of the intermediate data to local disk saves network bandwidth. Third, redundant execution can be used to reduce the impact of slow machines, and to handle machine failures and data loss.

Acknowledgements

Josh Levenberg has been instrumental in revising and extending the user-level MapReduce API with a number of new features based on his experience with using MapReduce and other people's suggestions for enhancements. MapReduce reads its input from and writes its output to the Google File System [8]. We would like to thank Mohit Aron, Howard Gobioff, Markus Gutschke, David Kramer, Shun-Tak Leung, and Josh Redstone for their work in developing GFS. We would also like to thank Percy Liang and Olcan Sercinoglu for their work in developing the cluster management system used by MapReduce. Mike Burrows, Wilson Hsieh, Josh Levenberg, Sharon Perl, Rob Pike, and Debby Wallach provided helpful comments on earlier drafts of this paper. The anonymous OSDI reviewers, and our shepherd, Eric Brewer, provided many useful suggestions of areas where the paper could be improved. Finally, we thank all the users of MapReduce within Google's engineering organization for providing helpful feedback, suggestions, and bug reports.

References

[1] Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, David E. Culler, Joseph M. Hellerstein, and David A. Patterson. High-performance sorting on networks of workstations. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, Tucson, Arizona, May 1997.

[2] Remzi H. Arpaci-Dusseau, Eric Anderson, Noah Treuhaft, David E. Culler, Joseph M. Hellerstein, David Patterson, and Kathy Yelick. Cluster I/O with River: Making the fast case common. In *Proceedings of the Sixth Workshop on Input/Output in Parallel and Distributed Systems (IOPADS '99)*, pages 10–22, Atlanta, Georgia, May 1999.

[3] Arash Baratloo, Mehmet Karaul, Zvi Kedem, and Peter Wyckoff. Charlotte: Metacomputing on the web. In *Proceedings of the 9th International Conference on Parallel and Distributed Computing Systems*, 1996.

[4] Luiz A. Barroso, Jeffrey Dean, and Urs Hölzle. Web search for a planet: The Google cluster architecture. *IEEE Micro*, 23(2):22–28, April 2003.

[5] John Bent, Douglas Thain, Andrea C.Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Miron Livny. Explicit control in a batch-aware distributed file system. In *Proceedings of the 1st USENIX Symposium on Networked Systems Design and Implementation NSDI*, March 2004.

[6] Guy E. Blelloch. Scans as primitive parallel operations. *IEEE Transactions on Computers*, C-38(11), November 1989.

[7] Armando Fox, Steven D. Gribble, Yatin Chawathe, Eric A. Brewer, and Paul Gauthier. Cluster-based scalable network services. In *Proceedings of the 16th ACM Symposium on Operating System Principles*, pages 78–91, Saint-Malo, France, 1997.

[8] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *19th Symposium on Operating Systems Principles*, pages 29–43, Lake George, New York, 2003.

[9] S. Gorlatch. Systematic efficient parallelization of scan and other list homomorphisms. In L. Bouge, P. Fraigniaud, A. Mignotte, and Y. Robert, editors, *Euro-Par'96. Parallel Processing*, Lecture Notes in Computer Science 1124, pages 401–408. Springer-Verlag, 1996.

[10] Jim Gray. Sort benchmark home page. http://research.microsoft.com/barc/SortBenchmark/.

[11] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press, Cambridge, MA, 1999.

[12] L. Huston, R. Sukthankar, R. Wickremesinghe, M. Satyanarayanan, G. R. Ganger, E. Riedel, and A. Ailamaki. Diamond: A storage architecture for early discard in interactive search. In *Proceedings of the 2004 USENIX File and Storage Technologies FAST Conference*, April 2004.

[13] Richard E. Ladner and Michael J. Fischer. Parallel prefix computation. *Journal of the ACM*, 27(4):831–838, 1980.

[14] Michael O. Rabin. Efficient dispersal of information for security, load balancing and fault tolerance. *Journal of the ACM*, 36(2):335–348, 1989.

[15] Erik Riedel, Christos Faloutsos, Garth A. Gibson, and David Nagle. Active disks for large-scale data processing. *IEEE Computer*, pages 68–74, June 2001.

[16] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice: The Condor experience. *Concurrency and Computation: Practice and Experience*, 2004.

[17] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1997.

[18] Jim Wyllie. Spsort: How to sort a terabyte quickly. http://alme1.almaden.ibm.com/cs/spsort.pdf.

A Word Frequency

This section contains a program that counts the number of occurrences of each unique word in a set of input files specified on the command line.

```
#include "mapreduce/mapreduce.h"

// User's map function
class WordCounter : public Mapper {
 public:
  virtual void Map(const MapInput& input) {
    const string& text = input.value();
    const int n = text.size();
    for (int i = 0; i < n; ) {
      // Skip past leading whitespace
      while ((i < n) && isspace(text[i]))
        i++;

      // Find word end
      int start = i;
      while ((i < n) && !isspace(text[i]))
        i++;

      if (start < i)
        Emit(text.substr(start,i-start),"1");
    }
  }
};
REGISTER_MAPPER(WordCounter);

// User's reduce function
class Adder : public Reducer {
  virtual void Reduce(ReduceInput* input) {
    // Iterate over all entries with the
    // same key and add the values
    int64 value = 0;
    while (!input->done()) {
      value += StringToInt(input->value());
      input->NextValue();
    }

    // Emit sum for input->key()
    Emit(IntToString(value));
  }
};
REGISTER_REDUCER(Adder);

int main(int argc, char** argv) {
  ParseCommandLineFlags(argc, argv);

  MapReduceSpecification spec;

  // Store list of input files into "spec"
  for (int i = 1; i < argc; i++) {
    MapReduceInput* input = spec.add_input();
    input->set_format("text");
    input->set_filepattern(argv[i]);
    input->set_mapper_class("WordCounter");
  }

  // Specify the output files:
  //    /gfs/test/freq-00000-of-00100
  //    /gfs/test/freq-00001-of-00100
  //    ...
  MapReduceOutput* out = spec.output();
  out->set_filebase("/gfs/test/freq");
  out->set_num_tasks(100);
  out->set_format("text");
  out->set_reducer_class("Adder");

  // Optional: do partial sums within map
  // tasks to save network bandwidth
  out->set_combiner_class("Adder");

  // Tuning parameters: use at most 2000
  // machines and 100 MB of memory per task
  spec.set_machines(2000);
  spec.set_map_megabytes(100);
  spec.set_reduce_megabytes(100);

  // Now run it
  MapReduceResult result;
  if (!MapReduce(spec, &result)) abort();

  // Done: 'result' structure contains info
  // about counters, time taken, number of
  // machines used, etc.

  return 0;
}
```

Chapter 14

Cassandra - A Decentralized Structured Storage System

Authors:
Avinash Lakshman, Prashant Malik

Publication year: 2010

Cassandra - A Decentralized Structured Storage System

Avinash Lakshman
Facebook

Prashant Malik
Facebook

ABSTRACT

Cassandra is a distributed storage system for managing very large amounts of structured data spread out across many commodity servers, while providing highly available service with no single point of failure. Cassandra aims to run on top of an infrastructure of hundreds of nodes (possibly spread across different data centers). At this scale, small and large components fail continuously. The way Cassandra manages the persistent state in the face of these failures drives the reliability and scalability of the software systems relying on this service. While in many ways Cassandra resembles a database and shares many design and implementation strategies therewith, Cassandra does not support a full relational data model; instead, it provides clients with a simple data model that supports dynamic control over data layout and format. Cassandra system was designed to run on cheap commodity hardware and handle high write throughput while not sacrificing read efficiency.

1. INTRODUCTION

Facebook runs the largest social networking platform that serves hundreds of millions users at peak times using tens of thousands of servers located in many data centers around the world. There are strict operational requirements on Facebook's platform in terms of performance, reliability and efficiency, and to support *continuous growth* the platform needs to be highly scalable. Dealing with failures in an infrastructure comprised of thousands of components is our standard mode of operation; there are always a small but significant number of server and network components that are failing at any given time. As such, the software systems need to be constructed in a manner that treats failures as the norm rather than the exception. To meet the reliability and scalability needs described above Facebook has developed Cassandra.

Cassandra uses a synthesis of well known techniques to achieve scalability and availability. Cassandra was designed to fulfill the storage needs of the Inbox Search problem. Inbox Search is a feature that enables users to search through their Facebook Inbox. At Facebook this meant the system was required to handle a very high write throughput, billions of writes per day, and also scale with the number of users. Since users are served from data centers that are geographically distributed, being able to replicate data across data centers was key to keep search latencies down. Inbox Search was launched in June of 2008 for around 100 million users and today we are at over 250 million users and Cassandra has kept up the promise so far. Cassandra is now deployed as the backend storage system for multiple services within Facebook.

This paper is structured as follows. Section 2 talks about related work, some of which has been very influential on our design. Section 3 presents the data model in more detail. Section 4 presents the overview of the client API. Section 5 presents the system design and the distributed algorithms that make Cassandra work. Section 6 details the experiences of making Cassandra work and refinements to improve performance. In Section 6.1 we describe how one of the applications in the Facebook platform uses Cassandra. Finally Section 7 concludes with future work on Cassandra.

2. RELATED WORK

Distributing data for performance, availability and durability has been widely studied in the file system and database communities. Compared to P2P storage systems that only support flat namespaces, distributed file systems typically support hierarchical namespaces. Systems like Ficus[14] and Coda[16] replicate files for high availability at the expense of consistency. Update conflicts are typically managed using specialized conflict resolution procedures. Farsite[2] is a distributed file system that does not use any centralized server. Farsite achieves high availability and scalability using replication. The Google File System (GFS)[9] is another distributed file system built for hosting the state of Google's internal applications. GFS uses a simple design with a single master server for hosting the entire metadata and where the data is split into chunks and stored in chunk servers. However the GFS master is now made fault tolerant using the Chubby[3] abstraction. Bayou[18] is a distributed relational database system that allows disconnected operations and provides eventual data consistency. Among these systems, Bayou, Coda and Ficus allow disconnected operations and are resilient to issues such as network partitions and outages. These systems differ on their conflict resolution procedures. For instance, Coda and Ficus perform system level conflict resolution and Bayou allows application level

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...$10.00.

resolution. All of them however, guarantee eventual consistency. Similar to these systems, Dynamo[6] allows read and write operations to continue even during network partitions and resolves update conflicts using different conflict resolution mechanisms, some client driven. Traditional replicated relational database systems focus on the problem of guaranteeing strong consistency of replicated data. Although strong consistency provides the application writer a convenient programming model, these systems are limited in scalability and availability [10]. These systems are not capable of handling network partitions because they typically provide strong consistency guarantees.

Dynamo[6] is a storage system that is used by Amazon to store and retrieve user shopping carts. Dynamo's Gossip based membership algorithm helps every node maintain information about every other node. Dynamo can be defined as a structured overlay with at most one-hop request routing. Dynamo detects updated conflicts using a vector clock scheme, but prefers a client side conflict resolution mechanism. A write operation in Dynamo also requires a read to be performed for managing the vector timestamps. This is can be very limiting in environments where systems need to handle a very high write throughput. Bigtable[4] provides both structure and data distribution but relies on a distributed file system for its durability.

3. DATA MODEL

A table in Cassandra is a distributed multi dimensional map indexed by a key. The value is an object which is highly structured. The row key in a table is a string with no size restrictions, although typically 16 to 36 bytes long. Every operation under a single row key is atomic per replica no matter how many columns are being read or written into. Columns are grouped together into sets called column families very much similar to what happens in the Bigtable[4] system. Cassandra exposes two kinds of columns families, Simple and Super column families. Super column families can be visualized as a column family within a column family.

Furthermore, applications can specify the sort order of columns within a Super Column or Simple Column family. The system allows columns to be sorted either by time or by name. Time sorting of columns is exploited by application like Inbox Search where the results are always displayed in time sorted order. Any column within a column family is accessed using the convention $column_family : column$ and any column within a column family that is of type super is accessed using the convention $column_family : super_column : column$. A very good example of the super column family abstraction power is given in Section 6.1. Typically applications use a dedicated Cassandra cluster and manage them as part of their service. Although the system supports the notion of multiple tables all deployments have only one table in their schema.

4. API

The Cassandra API consists of the following three simple methods.

- $insert(table, key, rowMutation)$
- $get(table, key, columnName)$
- $delete(table, key, columnName)$

$columnName$ can refer to a specific column within a column family, a column family, a super column family, or a column within a super column.

5. SYSTEM ARCHITECTURE

The architecture of a storage system that needs to operate in a production setting is complex. In addition to the actual data persistence component, the system needs to have the following characteristics; scalable and robust solutions for load balancing, membership and failure detection, failure recovery, replica synchronization, overload handling, state transfer, concurrency and job scheduling, request marshalling, request routing, system monitoring and alarming, and configuration management. Describing the details of each of the solutions is beyond the scope of this paper, so we will focus on the core distributed systems techniques used in Cassandra: partitioning, replication, membership, failure handling and scaling. All these modules work in synchrony to handle read/write requests. Typically a read/write request for a key gets routed to any node in the Cassandra cluster. The node then determines the replicas for this particular key. For writes, the system routes the requests to the replicas and waits for a quorum of replicas to acknowledge the completion of the writes. For reads, based on the consistency guarantees required by the client, the system either routes the requests to the closest replica or routes the requests to all replicas and waits for a quorum of responses.

5.1 Partitioning

One of the key design features for Cassandra is the ability to scale incrementally. This requires, the ability to dynamically partition the data over the set of nodes (i.e., storage hosts) in the cluster. Cassandra partitions data across the cluster using consistent hashing [11] but uses an order preserving hash function to do so. In consistent hashing the output range of a hash function is treated as a fixed circular space or "ring" (i.e. the largest hash value wraps around to the smallest hash value). Each node in the system is assigned a random value within this space which represents its *position* on the ring. Each data item identified by a key is assigned to a node by hashing the data item's key to yield its position on the ring, and then walking the ring clockwise to find the first node with a position larger than the item's position. This node is deemed the coordinator for this key. The application specifies this key and the Cassandra uses it to route requests. Thus, each node becomes responsible for the region in the ring between it and its predecessor node on the ring. The principal advantage of consistent hashing is that departure or arrival of a node only affects its immediate neighbors and other nodes remain unaffected. The basic consistent hashing algorithm presents some challenges. First, the random position assignment of each node on the ring leads to non-uniform data and load distribution. Second, the basic algorithm is oblivious to the heterogeneity in the performance of nodes. Typically there exist two ways to address this issue: One is for nodes to get assigned to multiple positions in the circle (like in Dynamo), and the second is to analyze load information on the ring and have lightly loaded nodes move on the ring to alleviate heavily loaded nodes as described in [17]. Cassandra opts for the latter as it makes the design and implementation very tractable and helps to make very deterministic choices about load balancing.

5.2 Replication

Cassandra uses replication to achieve high availability and durability. Each data item is replicated at N hosts, where N is the replication factor configured "per-instance". Each key, k, is assigned to a coordinator node (described in the previous section). The coordinator is in charge of the replication of the data items that fall within its range. In addition to locally storing each key within its range, the coordinator replicates these keys at the N-1 nodes in the ring. Cassandra provides the client with various options for how data needs to be replicated. Cassandra provides various replication policies such as "Rack Unaware", "Rack Aware" (within a datacenter) and "Datacenter Aware". Replicas are chosen based on the replication policy chosen by the application. If certain application chooses "Rack Unaware" replication strategy then the non-coordinator replicas are chosen by picking N-1 successors of the coordinator on the ring. For "Rack Aware" and "Datacenter Aware" strategies the algorithm is slightly more involved. Cassandra system elects a leader amongst its nodes using a system called Zookeeper[13]. All nodes on joining the cluster contact the leader who tells them for what ranges they are replicas for and leader makes a concerted effort to maintain the invariant that no node is responsible for more than N-1 ranges in the ring. The metadata about the ranges a node is responsible is cached locally at each node and in a fault-tolerant manner inside Zookeeper - this way a node that crashes and comes back up knows what ranges it was responsible for. We borrow from Dynamo parlance and deem the nodes that are responsible for a given range the "preference list" for the range.

As is explained in Section 5.1 every node is aware of every other node in the system and hence the range they are responsible for. Cassandra provides durability guarantees in the presence of node failures and network partitions by relaxing the quorum requirements as described in Section5.2. Data center failures happen due to power outages, cooling failures, network failures, and natural disasters. Cassandra is configured such that each row is replicated across multiple data centers. In essence, the preference list of a key is constructed such that the storage nodes are spread across multiple datacenters. These datacenters are connected through high speed network links. This scheme of replicating across multiple datacenters allows us to handle entire data center failures without any outage.

5.3 Membership

Cluster membership in Cassandra is based on Scuttlebutt[19], a very efficient anti-entropy Gossip based mechanism. The salient feature of Scuttlebutt is that it has very efficient CPU utilization and very efficient utilization of the gossip channel. Within the Cassandra system Gossip is not only used for membership but also to disseminate other system related control state.

5.3.1 Failure Detection

Failure detection is a mechanism by which a node can locally determine if any other node in the system is up or down. In Cassandra failure detection is also used to avoid attempts to communicate with unreachable nodes during various operations. Cassandra uses a modified version of the Φ Accrual Failure Detector[8]. The idea of an Accrual Failure Detection is that the failure detection module doesn't emit a Boolean value stating a node is up or down. Instead the failure detection module emits a value which represents a suspicion level for each of monitored nodes. This value is defined as Φ. The basic idea is to express the value of Φ on a scale that is dynamically adjusted to reflect network and load conditions at the monitored nodes.

Φ has the following meaning: Given some threshold Φ, and assuming that we decide to suspect a node A when $\Phi = 1$, then the likelihood that we will make a mistake (i.e., the decision will be contradicted in the future by the reception of a late heartbeat) is about 10%. The likelihood is about 1% with $\Phi = 2$, 0.1% with $\Phi = 3$, and so on. Every node in the system maintains a sliding window of inter-arrival times of gossip messages from other nodes in the cluster. The distribution of these inter-arrival times is determined and Φ is calculated. Although the original paper suggests that the distribution is approximated by the Gaussian distribution we found the Exponential Distribution to be a better approximation, because of the nature of the gossip channel and its impact on latency. To our knowledge our implementation of the Accrual Failure Detection in a Gossip based setting is the first of its kind. Accrual Failure Detectors are very good in both their accuracy and their speed and they also adjust well to network conditions and server load conditions.

5.4 Bootstrapping

When a node starts for the first time, it chooses a random token for its position in the ring. For fault tolerance, the mapping is persisted to disk locally and also in Zookeeper. The token information is then gossiped around the cluster. This is how we know about all nodes and their respective positions in the ring. This enables any node to route a request for a key to the correct node in the cluster. In the bootstrap case, when a node needs to join a cluster, it reads its configuration file which contains a list of a few contact points within the cluster. We call these initial contact points, seeds of the cluster. Seeds can also come from a configuration service like Zookeeper.

In Facebook's environment node outages (due to failures and maintenance tasks) are often transient but may last for extended intervals. Failures can be of various forms such as disk failures, bad CPU etc. A node outage rarely signifies a permanent departure and therefore should not result in re-balancing of the partition assignment or repair of the unreachable replicas. Similarly, manual error could result in the unintentional startup of new Cassandra nodes. To that effect every message contains the cluster name of each Cassandra instance. If a manual error in configuration led to a node trying to join a wrong Cassandra instance it can thwarted based on the cluster name. For these reasons, it was deemed appropriate to use an explicit mechanism to initiate the addition and removal of nodes from a Cassandra instance. An administrator uses a command line tool or a browser to connect to a Cassandra node and issue a membership change to join or leave the cluster.

5.5 Scaling the Cluster

When a new node is added into the system, it gets assigned a token such that it can alleviate a heavily loaded node. This results in the new node splitting a range that some other node was previously responsible for. The Cassandra bootstrap algorithm is initiated from any other node in the system by an operator using either a command line utility

or the Cassandra web dashboard. The node giving up the data streams the data over to the new node using kernel-kernel copy techniques. Operational experience has shown that data can be transferred at the rate of 40 MB/sec from a single node. We are working on improving this by having multiple replicas take part in the bootstrap transfer thereby parallelizing the effort, similar to Bittorrent.

5.6 Local Persistence

The Cassandra system relies on the local file system for data persistence. The data is represented on disk using a format that lends itself to efficient data retrieval. Typical write operation involves a write into a commit log for durability and recoverability and an update into an in-memory data structure. The write into the in-memory data structure is performed only after a successful write into the commit log. We have a dedicated disk on each machine for the commit log since all writes into the commit log are sequential and so we can maximize disk throughput. When the in-memory data structure crosses a certain threshold, calculated based on data size and number of objects, it dumps itself to disk. This write is performed on one of many commodity disks that machines are equipped with. All writes are sequential to disk and also generate an index for efficient lookup based on row key. These indices are also persisted along with the data file. Over time many such files could exist on disk and a merge process runs in the background to collate the different files into one file. This process is very similar to the compaction process that happens in the Bigtable system.

A typical read operation first queries the in-memory data structure before looking into the files on disk. The files are looked at in the order of newest to oldest. When a disk lookup occurs we could be looking up a key in multiple files on disk. In order to prevent lookups into files that do not contain the key, a bloom filter, summarizing the keys in the file, is also stored in each data file and also kept in memory. This bloom filter is first consulted to check if the key being looked up does indeed exist in the given file. A key in a column family could have many columns. Some special indexing is required to retrieve columns which are further away from the key. In order to prevent scanning of every column on disk we maintain column indices which allow us to jump to the right chunk on disk for column retrieval. As the columns for a given key are being serialized and written out to disk we generate indices at every 256K chunk boundary. This boundary is configurable, but we have found 256K to work well for us in our production workloads.

5.7 Implementation Details

The Cassandra process on a single machine is primarily consists of the following abstractions: partitioning module, the cluster membership and failure detection module and the storage engine module. Each of these modules rely on an event driven substrate where the message processing pipeline and the task pipeline are split into multiple stages along the line of the SEDA[20] architecture. Each of these modules has been implemented from the ground up using Java. The cluster membership and failure detection module, is built on top of a network layer which uses non-blocking I/O. All system control messages rely on UDP based messaging while the application related messages for replication and request routing relies on TCP. The request routing modules are implemented using a certain state machine. When a read/write request arrives at any node in the cluster the state machine morphs through the following states (i) identify the node(s) that own the data for the key (ii) route the requests to the nodes and wait on the responses to arrive (iii) if the replies do not arrive within a configured timeout value fail the request and return to the client (iv) figure out the latest response based on timestamp (v) schedule a repair of the data at any replica if they do not have the latest piece of data. For sake of exposition we do not talk about failure scenarios here. The system can be configured to perform either synchronous or asynchronous writes. For certain systems that require high throughput we rely on asynchronous replication. Here the writes far exceed the reads that come into the system. During the synchronous case we wait for a quorum of responses before we return a result to the client.

In any journaled system there needs to exist a mechanism for purging commit log entries. In Cassandra we use a rolling a commit log where a new commit log is rolled out after an older one exceeds a particular, configurable, size. We have found that rolling commit logs after 128MB size seems to work very well in our production workloads. Every commit log has a header which is basically a bit vector whose size is fixed and typically more than the number of column families that a particular system will ever handle. In our implementation we have an in-memory data structure and a data file that is generated per column family. Every time the in-memory data structure for a particular column family is dumped to disk we set its bit in the commit log stating that this column family has been successfully persisted to disk. This is an indication that this piece of information is already committed. These bit vectors are per commit log and also maintained in memory. Every time a commit log is rolled its bit vector and all the bit vectors of commit logs rolled prior to it are checked. If it is deemed that all the data has been successfully persisted to disk then these commit logs are deleted. The write operation into the commit log can either be in normal mode or in *fast sync* mode. In the fast sync mode the writes to the commit log are buffered. This implies that there is a potential of data loss on machine crash. In this mode we also dump the in-memory data structure to disk in a buffered fashion. Traditional databases are not designed to handle particularly high write throughput. Cassandra morphs all writes to disk into sequential writes thus maximizing disk write throughput. Since the files dumped to disk are never mutated no locks need to be taken while reading them. The server instance of Cassandra is practically lockless for read/write operations. Hence we do not need to deal with or handle the concurrency issues that exist in B-Tree based database implementations.

The Cassandra system indexes all data based on primary key. The data file on disk is broken down into a sequence of blocks. Each block contains at most 128 keys and is demarcated by a block index. The block index captures the relative offset of a key within the block and the size of its data. When an in-memory data structure is dumped to disk a block index is generated and their offsets written out to disk as indices. This index is also maintained in memory for fast access. A typical read operation always looks up data first in the in-memory data structure. If found the data is returned to the application since the in-memory data structure contains the latest data for any key. If not found then we perform disk I/O against all the data files on disk in reverse time order. Since we are always looking for the latest

data we look into the latest file first and return if we find the data. Over time the number of data files will increase on disk. We perform a compaction process, very much like the Bigtable system, which merges multiple files into one; essentially merge sort on a bunch of sorted data files. The system will always compact files that are close to each other with respect to size i.e there will never be a situation where a 100GB file is compacted with a file which is less than 50GB. Periodically a major compaction process is run to compact all related data files into one big file. This compaction process is a disk I/O intensive operation. Many optimizations can be put in place to not affect in coming read requests.

6. PRACTICAL EXPERIENCES

In the process of designing, implementing and maintaining Cassandra we gained a lot of useful experience and learned numerous lessons. One very fundamental lesson learned was not to add any new feature without understanding the effects of its usage by applications. Most problematic scenarios do not stem from just node crashes and network partitions. We share just a few interesting scenarios here.

- Before launching the Inbox Search application we had to index 7TB of inbox data for over 100M users, then stored in our MySQL[1] infrastructure, and load it into the Cassandra system. The whole process involved running Map/Reduce[7] jobs against the MySQL data files, indexing them and then storing the reverse-index in Cassandra. The M/R process actually behaves as the client of Cassandra. We exposed some background channels for the M/R process to aggregate the reverse index per user and send over the serialized data over to the Cassandra instance, to avoid the serialization/deserialization overhead. This way the Cassandra instance is only bottlenecked by network bandwidth.

- Most applications only require atomic operation per key per replica. However there have been some applications that have asked for transactional mainly for the purpose of maintaining secondary indices. Most developers with years of development experience working with RDBMS's find this a very useful feature to have. We are working on a mechanism to expose such atomic operations.

- We experimented with various implementations of Failure Detectors such as the ones described in [15] and [5]. Our experience had been that the time to detect failures increased beyond an acceptable limit as the size of the cluster grew. In one particular experiment in a cluster of 100 nodes time to taken to detect a failed node was in the order of two minutes. This is practically unworkable in our environments. With the accrual failure detector with a slightly conservative value of PHI, set to 5, the average time to detect failures in the above experiment was about 15 seconds.

- Monitoring is not to be taken for granted. The Cassandra system is well integrated with Ganglia[12], a distributed performance monitoring tool. We expose various system level metrics to Ganglia and this has helped us understand the behavior of the system when subject to our production workload. Disks fail for no apparent reasons. The bootstrap algorithm has some hooks to repair nodes when disk fail. This is however an administrative operation.

- Although Cassandra is a completely decentralized system we have learned that having some amount of coordination is essential to making the implementation of some distributed features tractable. For example Cassandra is integrated with Zookeeper, which can be used for various coordination tasks in large scale distributed systems. We intend to use the Zookeeper abstraction for some key features which actually do not come in the way of applications that use Cassandra as the storage engine.

6.1 Facebook Inbox Search

For Inbox Search we maintain a per user index of all messages that have been exchanged between the sender and the recipients of the message. There are two kinds of search features that are enabled today (a) term search (b) interactions - given the name of a person return all messages that the user might have ever sent or received from that person. The schema consists of two column families. For query (a) the user id is the key and the words that make up the message become the super column. Individual message identifiers of the messages that contain the word become the columns within the super column. For query (b) again the user id is the key and the recipients id's are the super columns. For each of these super columns the individual message identifiers are the columns. In order to make the searches fast Cassandra provides certain hooks for intelligent caching of data. For instance when a user clicks into the search bar an asynchronous message is sent to the Cassandra cluster to prime the buffer cache with that user's index. This way when the actual search query is executed the search results are likely to already be in memory. The system currently stores about 50+TB of data on a 150 node cluster, which is spread out between east and west coast data centers. We show some production measured numbers for read performance.

Latency Stat	Search Interactions	Term Search
Min	7.69ms	7.78ms
Median	15.69ms	18.27ms
Max	26.13ms	44.41ms

7. CONCLUSION

We have built, implemented, and operated a storage system providing scalability, high performance, and wide applicability. We have empirically demonstrated that Cassandra can support a very high update throughput while delivering low latency. Future works involves adding compression, ability to support atomicity across keys and secondary index support.

8. ACKNOWLEDGEMENTS

Cassandra system has benefitted greatly from feedback from many individuals within Facebook. In addition we thank Karthik Ranganathan who indexed all the existing data in MySQL and moved it into Cassandra for our first production deployment. We would also like to thank Dan Dumitriu from EPFL for his valuable suggestions about [19] and [8].

9. REFERENCES

[1] MySQL AB. Mysql.

[2] Atul Adya, William J. Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R. Douceur, Jon Howell, Jacob R. Lorch, Marvin Theimer, and Roger P. Wattenhofer. Farsite: Federated, available, and reliable storage for an incompletely trusted environment. In *In Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI*, pages 1–14, 2002.

[3] Mike Burrows. The chubby lock service for loosely-coupled distributed systems. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, pages 335–350, Berkeley, CA, USA, 2006. USENIX Association.

[4] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. In *In Proceedings of the 7th Conference on USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, pages 205–218, 2006.

[5] Abhinandan Das, Indranil Gupta, and Ashish Motivala. Swim: Scalable weakly-consistent infection-style process group membership protocol. In *DSN '02: Proceedings of the 2002 International Conference on Dependable Systems and Networks*, pages 303–312, Washington, DC, USA, 2002. IEEE Computer Society.

[6] Giuseppe de Candia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazonÕs highly available key-value store. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 205–220. ACM, 2007.

[7] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[8] Xavier Défago, Péter Urbán, Naohiro Hayashibara, and Takuya Katayama. The ϕ accrual failure detector. In *RR IS-RR-2004-010, Japan Advanced Institute of Science and Technology*, pages 66–78, 2004.

[9] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 29–43, New York, NY, USA, 2003. ACM.

[10] Jim Gray and Pat Helland. The dangers of replication and a solution. In *In Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 173–182, 1996.

[11] David Karger, Eric Lehman, Tom Leighton, Matthew Levine, Daniel Lewin, and Rina Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *In ACM Symposium on Theory of Computing*, pages 654–663, 1997.

[12] Matthew L. Massie, Brent N. Chun, and David E. Culler. The ganglia distributed monitoring system: Design, implementation, and experience. *Parallel Computing*, 30:2004, 2004.

[13] Benjamin Reed and Flavio Junquieira. Zookeeper.

[14] Peter Reiher, John Heidemann, David Ratner, Greg Skinner, and Gerald Popek. Resolving file conflicts in the ficus file system. In *USTC'94: Proceedings of the USENIX Summer 1994 Technical Conference on USENIX Summer 1994 Technical Conference*, pages 12–12, Berkeley, CA, USA, 1994. USENIX Association.

[15] Robbert Van Renesse, Yaron Minsky, and Mark Hayden. A gossip-style failure detection service. In *Service,T̃ Proc. Conf. Middleware*, pages 55–70, 1996.

[16] Mahadev Satyanarayanan, James J. Kistler, Puneet Kumar, Maria E. Okasaki, Ellen H. Siegel, and David C. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Trans. Comput.*, 39(4):447–459, 1990.

[17] Ion Stoica, Robert Morris, David Liben-nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Transactions on Networking*, 11:17–32, 2003.

[18] D. B. Terry, M. M. Theimer, Karin Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. In *SOSP '95: Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 172–182, New York, NY, USA, 1995. ACM.

[19] Robbert van Renesse, Dan Mihai Dumitriu, Valient Gough, and Chris Thomas. Efficient reconciliation and flow control for anti-entropy protocols. In *Proceedings of the 2nd Large Scale Distributed Systems and Middleware Workshop (LADIS '08)*, New York, NY, USA, 2008. ACM.

[20] Matt Welsh, David Culler, and Eric Brewer. Seda: an architecture for well-conditioned, scalable internet services. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 230–243, New York, NY, USA, 2001. ACM.

Chapter 15

Paxos Made Live - An Engineering Perspective

Authors:
Tushar Chandra, Robert Griesemer, Joshua Redstone

Publication year: 2007

Paxos Made Live - An Engineering Perspective

Tushar Chandra
Robert Griesemer
Joshua Redstone

June 20, 2007

Abstract

We describe our experience in building a fault-tolerant data-base using the Paxos consensus algorithm. Despite the existing literature in the field, building such a database proved to be non-trivial. We describe selected algorithmic and engineering problems encountered, and the solutions we found for them. Our measurements indicate that we have built a competitive system.

1 Introduction

It is well known that fault-tolerance on commodity hardware can be achieved through replication [17, 18]. A common approach is to use a consensus algorithm [7] to ensure that all replicas are mutually consistent [8, 14, 17]. By repeatedly applying such an algorithm on a sequence of input values, it is possible to build an identical log of values on each replica. If the values are operations on some data structure, application of the same log on all replicas may be used to arrive at mutually consistent data structures on all replicas. For instance, if the log contains a sequence of database operations, and if the same sequence of operations is applied to the (local) database on each replica, eventually all replicas will end up with the same database content (provided that they all started with the same initial database state).

This general approach can be used to implement a wide variety of fault-tolerant primitives, of which a fault-tolerant database is just an example. As a result, the consensus problem has been studied extensively over the past two decades. There are several well-known consensus algorithms that operate within a multitude of settings and which tolerate a variety of failures. The Paxos consensus algorithm [8] has been discussed in the theoretical [16] and applied community [10, 11, 12] for over a decade.

We used the Paxos algorithm ("Paxos") as the base for a framework that implements a fault-tolerant log. We then relied on that framework to build a fault-tolerant database. Despite the existing literature on the subject, building a production system turned out to be a non-trivial task for a variety of reasons:

- While Paxos can be described with a page of pseudo-code, our complete implementation contains several thousand lines of C++ code. The blow-up is not due simply to the fact that we used C++ instead of pseudo notation, nor because our code style may have been verbose. Converting the algorithm into a practical, production-ready system involved implementing many features and optimizations – some published in the literature and some not.

- The fault-tolerant algorithms community is accustomed to proving short algorithms (one page of pseudo code) correct. This approach does not scale to a system with thousands of lines of code. To gain confidence in the "correctness" of a real system, different methods had to be used.

- Fault-tolerant algorithms tolerate a limited set of carefully selected faults. However, the real world exposes software to a wide variety of failure modes, including errors in the algorithm, bugs in its

©ACM 2007. This is a minor revision of the work that will be published in the proceedings of ACM PODC 2007.

implementation, and operator error. We had to engineer the software and design operational procedures to robustly handle this wider set of failure modes.

- A real system is rarely specified precisely. Even worse, the specification may change during the implementation phase. Consequently, an implementation should be malleable. Finally, a system might "fail" due to a misunderstanding that occurred during its specification phase.

This paper discusses a selection of the algorithmic and engineering challenges we encountered in moving Paxos from theory to practice. This exercise took more R&D efforts than a straightforward translation of pseudo-code to C++ might suggest.

The rest of this paper is organized as follows. The next two sections expand on the motivation for this project and describe the general environment into which our system was built. We then provide a quick refresher on Paxos. We divide our experiences into three categories and discuss each in turn: algorithmic gaps in the literature, software engineering challenges, and unexpected failures. We conclude with measurements of our system, and some broader observations on the state of the art in our field.

2 Background

Chubby [1] is a fault-tolerant system at Google that provides a distributed locking mechanism and stores small files. Typically there is one Chubby instance, or "cell", per data center. Several Google systems – such as the Google Filesystem (GFS) [4] and Bigtable [2] – use Chubby for distributed coordination and to store a small amount of metadata.

Chubby achieves fault-tolerance through replication. A typical Chubby cell consists of five replicas, running the same code, each running on a dedicated machine. Every Chubby object (e.g., a Chubby lock, or file) is stored as an entry in a database. It is this database that is replicated. At any one time, one of these replicas is considered to be the "master".

Chubby clients (such as GFS and Bigtable) contact a Chubby cell for service. The master replica serves all Chubby requests. If a Chubby client contacts a replica that is not the master, the replica replies with the master's network address. The Chubby client may then contact the master. If the master fails, a new master is automatically elected, which will then continue to serve traffic based on the contents of its local copy of the replicated database. Thus, the replicated database ensures continuity of Chubby state across master failover.

The first version of Chubby was based on a commercial, third-party, fault-tolerant database; we will refer to this database as "3DB" for the rest of this paper. This database had a history of bugs related to replication. In fact, as far as we know, the replication mechanism was not based on a proven replication algorithm and we do not know if it is correct. Given the history of problems associated with that product and the importance of Chubby, we eventually decided to replace 3DB with our own solution based on the Paxos algorithm.

3 Architecture outline

Figure 1 illustrates the architecture of a single Chubby replica. A fault-tolerant replicated log based on the Paxos algorithm sits at the bottom of the protocol stack. Each replica maintains a local copy of the log. The Paxos algorithm is run repeatedly as required to ensure that all replicas have identical sequences of entries in their local logs. Replicas communicate with each other through a Paxos-specific protocol.

The next layer is a fault-tolerant replicated database which includes a local copy of the database at each replica. The database consists of a local *snapshot* and a *replay*-log of database operations. New database operations are submitted to the replicated log. When a database operation appears at a replica, it is applied on that replica's local database copy.

Finally, Chubby uses the fault-tolerant database to store its state. Chubby clients communicate with a single Chubby replica through a Chubby-specific protocol.

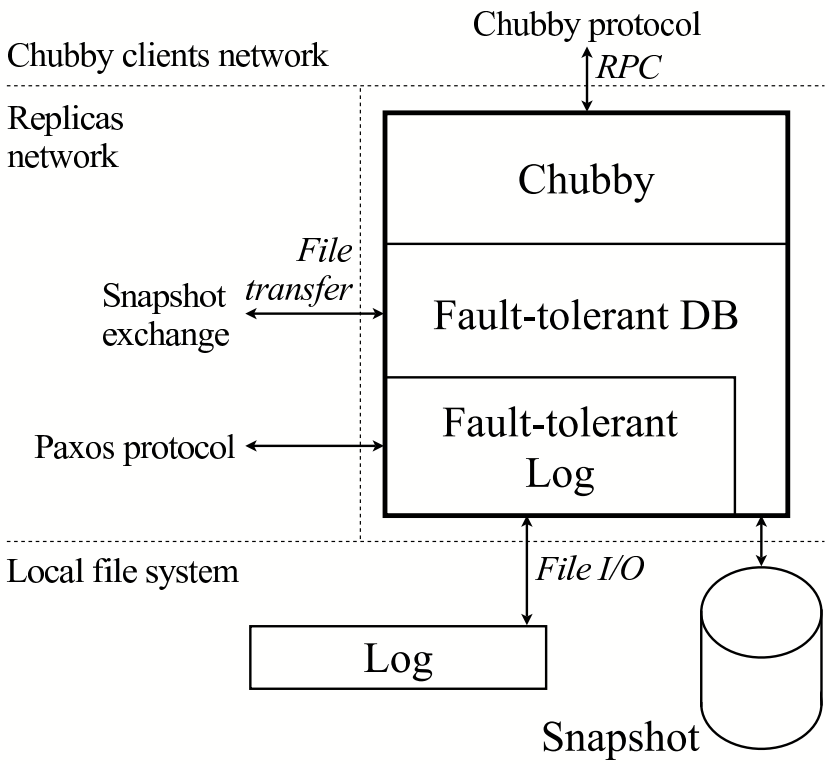


Figure 1: A single Chubby replica.

We devoted effort to designing clean interfaces separating the Paxos framework, the database, and Chubby. We did this partly for clarity while developing this system, but also with the intention of reusing the replicated log layer in other applications. We anticipate future systems at Google that seek fault-tolerance through replication. We believe that a fault-tolerant log is a powerful primitive on which to build such systems.

Our fault-tolerant log's API is depicted in Figure 2. It contains a call to *submit* a new value to the log. Once a submitted value enters the fault-tolerant log, our system invokes a callback to the client application at each replica and passes the submitted value.

Our system is multi-threaded and multiple values can be submitted concurrently on different threads. The replicated log does not create its own threads but can be invoked concurrently by any number of threads. This approach to threading helps in testing the system, as we will show in more detail later in the paper.

4 On Paxos

In this section we give an informal overview of the basic Paxos algorithm and outline how to chain together multiple executions of it (Multi-Paxos). We refer the reader to the literature for more formal descriptions and correctness proofs [8, 9, 16]. Readers who are familiar with Paxos may skip directly to the next section.

4.1 Paxos Basics

Paxos is a consensus algorithm executed by a set of processes, termed *replicas*, to agree on a single value in the presence of failures. Replicas may crash and subseqently recover. The network may drop messages

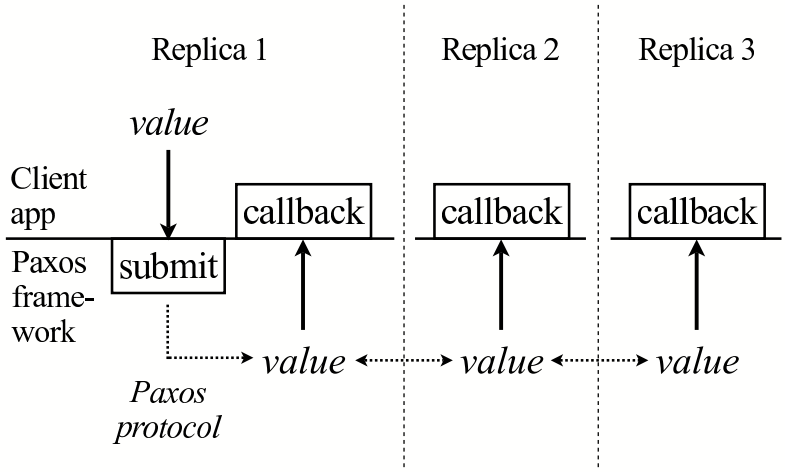


Figure 2: API for fault-tolerant log.

between replicas. Replicas have access to persistent storage that survives crashes. Some replicas may *submit* values for consensus. If eventually a majority of the replicas run for long enough without crashing and there are no failures, all running replicas are guaranteed to *agree* on one of the values that was submitted. In our system, the value to be agreed upon is the next entry in a (replicated) log as described in the introduction.

The algorithm consists of three phases, which may be repeated (because of failures):

1. Elect a replica to be the *coordinator*.
2. The coordinator selects a value and broadcasts it to all replicas in a message called the *accept* message. Other replicas either *acknowledge* this message or *reject* it.
3. Once a majority of the replicas *acknowledge* the coordinator, consensus has been reached, and the coordinator broadcasts a *commit* message to notify replicas.

To provide some intuition about how the algorithm works, consider first the case in which there is only a single coordinator and no failures. Consensus is reached once a majority of replicas receive the accept message from the coordinator and acknowledge it. Subsequently, if any minority of the replicas fail, we are still guaranteed that at least one replica will be alive that received the consensus value.

In reality the coordinator may fail. Paxos does not require that only one replica act as coordinator at a time. Multiple replicas may decide to become coordinators and execute the algorithm at any time. Typically the system is engineered to limit coordinator turnover, as it can delay reaching consensus.

This flexible election policy means there may be multiple replicas who simultaneously believe they are the coordinator. Further, these coordinators may select different values. Paxos ensures consensus can be reached on a single value (it can be from any coordinator) by introducing two extra mechanisms: 1) assigning an ordering to the successive coordinators; and 2) restricting each coordinator's choice in selecting a value.

Ordering the coordinators allows each replica to distinguish between the current coordinator and previous coordinators. In this way, replicas can reject messages from old coordinators and prevent them from disrupting consensus once it is reached. Paxos orders the coordinators by assigning them an increasing *sequence number* as follows. Each replica keeps track of the most recent sequence number it has seen so far. When a replica wants to become coordinator, it generates a unique[1] sequence number higher than any it has seen,

[1]For example, in a system with n replicas, assign each replica r a unique id i_r between 0 and $n-1$. Replica r picks the smallest sequence number s larger than any it has seen such that $s \ mod \ n = i_r$.

and broadcasts it to all replicas in a *propose* message. If a majority of replicas reply and indicate they have not seen a higher sequence number, then the replica acts as a coordinator. These replies are called *promise* messages since replicas promise henceforth to reject messages from old coordinators. This propose/promise message exchange constitutes step 1 listed above.

Once consensus is reached on a value, Paxos must force future coordinators to select that same value in order to ensure continued agreement. To guarantee this, the promise messages from replicas include the most recent value they have heard, if any, along with the sequence number of the coordinator from whom they heard it. The new coordinator chooses the value from the most recent coordinator. If none of the promise messages contain a value, the coordinator is free to choose a submitted value.

The reasoning why this works is subtle, but proceeds roughly as follows. The new coordinator requires a response to the propose message from a majority of replicas. Therefore, if consensus was achieved by a previous coordinator, the new coordinator is guaranteed to hear about the value decided upon from at least one replica. By induction, that value will have the highest sequence number of all responses received, and so will be selected by the new coordinator.

4.2 Multi-Paxos

Practical systems use Paxos as a building block to achieve consensus on a *sequence of values*, such as in a replicated log. The simple way to implement this is to repeatedly execute the Paxos algorithm. We term each execution an *instance* of Paxos. We refer to *submitting* a value to Paxos (or equivalently, to the log) to mean executing an instance of Paxos while submitting that value.

In Multi-Paxos some slow (*lagging*) replicas might not have participated in recent Paxos instances. We use a *catch-up* mechanism to enable lagging replicas to catch up with *leading* replicas.

Each replica maintains a locally persistent log to record all Paxos actions. When a replica crashes and subsequently recovers, it replays the persistent log to reconstruct its state prior to crashing. Replicas also use this log when helping lagging replicas to catch up. The Paxos algorithm as described thus far requires all message senders to log their state before sending messages – thus the algorithm requires a sequence of five writes (for each of the propose, promise, accept, acknowledgment, and commit messages) to disk on its critical path. Note that all writes have to be flushed to disk immediately before the system can proceed any further. In a system where replicas are in close network proximity, disk flush time can dominate the overall latency of the implementation.

There is a well-known optimization to reduce the number of messages involved by chaining together multiple Paxos instances [9]. Propose messages may be omitted if the coordinator identity does not change between instances. This does not interfere with the properties of Paxos because any replica at any time can still try to become coordinator by broadcasting a propose message with a higher sequence number. In order to avail itself of this optimization, a Multi-Paxos algorithm may be designed to pick a coordinator for long periods of time, trying not to let the coordinator change. We refer to this coordinator as the *master*. With this optimization, the Paxos algorithm only requires a single write to disk per Paxos instance on each replica, executed in parallel with each other. The master writes to disk immediately after sending its accept message and other replicas write to disk prior to sending their acknowledge message.

In order to get additional throughput in a concurrent system, it is possible to batch a collection of values submitted by different application threads into a single Paxos instance.

5 Algorithmic challenges

While the core Paxos algorithm is well-described, implementing a fault-tolerant log based on it is a non-trivial endeavor. Some of the complications are due to imperfections found in the real world (such as hard disk failures, or finite resources), and some are due to additional requirements (for instance, "master leases"). Many of these challenges have algorithmic solutions that are intimately connected with the core Paxos algorithm. In the following we describe a number of mechanisms that we introduced.

5.1 Handling disk corruption

Replicas witness disk corruption from time to time. A disk may be corrupted due to a media failure or due to an operator error (an operator may accidentally erase critical data). When a replica's disk is corrupted and it loses its persistent state, it may renege on promises it has made to other replicas in the past. This violates a key assumption in the Paxos algorithm. We use the following mechanism to address this problem [14].

Disk corruptions manifest themselves in two ways. Either file(s) contents may change or file(s) may become inaccessible. To detect the former, we store the checksum of the contents of each file in the file[2]. The latter may be indistinguishable from a new replica with an empty disk – we detect this case by having a new replica leave a marker in GFS after start-up. If this replica ever starts again with an empty disk, it will discover the GFS marker and indicate that it has a corrupted disk.

A replica with a corrupted disk rebuilds its state as follows. It participates in Paxos as a non-voting member; meaning that it uses the catch-up mechanism to catch up but does not respond with promise or acknowledgment messages. It remains in this state until it observes one complete instance of Paxos that was started after the replica started rebuilding its state. By waiting for the extra instance of Paxos, we ensure that this replica could not have reneged on an earlier promise.

This mechanism enables the following optimization to improve the latency of the system. Since the system can now deal with occasional disk corruption, under some circumstances it may be acceptable not to flush writes to disk immediately[3]. While we have considered schemes to exploit this observation, we have not implemented them yet.

5.2 Master leases

When the basic Paxos algorithm is used to implement a replicated data structure, reads of the data structure require executing an instance of Paxos. This serializes the read with respect to updates and ensures that the *current* state is read. In particular, read operations cannot be served out of the master's copy of the data structure because it is possible that other replicas have elected another master and modified the data structure without notifying the old master. In this case, the read operation at the master runs the risk of returning stale data. Since read operations usually comprise a large fraction of all operations, serializing reads through Paxos is expensive.

The workaround is to implement *master leases* [5] with the following semantics: as long as the master has the lease, it is guaranteed that other replicas cannot successfully submit values to Paxos. Thus a master with the lease has up-to-date information in its local data structure which can be used to serve a read operation purely locally. By making the master attempt to renew its lease before it expires we can ensure that a master has a lease most of the time. With our system, masters successfully maintain leases for several days at a time.

In our implementation, all replicas implicitly grant a lease to the master of the previous Paxos instance and refuse to process Paxos messages from any other replica while the lease is held. The master maintains a shorter timeout for the lease than the replicas – this protects the system against clock drift. The master periodically submits a dummy "heartbeat" value to Paxos to refresh its lease.

The Multi-Paxos optimization exhibits the following stability problem when there are intermittent network outages. When a master temporarily disconnects, Paxos will elect a new master. The new master will maintain a fixed sequence number across instances of Paxos. In the mean time, when the disconnected old master tries to run the Paxos algorithm, if it manages to connect with another replica, it may increase its sequence number. When it reconnects, it may have a higher sequence number than the new master and be able to replace the new master. Later it may disconnect again, and the cycle can repeat itself.

This behavior is undesirable as Chubby master changes have a negative impact on some of its users. Furthermore, this behavior can degenerate into rapid master changes in a network with poor connectivity.

[2]This mechanism will not detect files that roll back to an old state. We believe this is an unlikely scenario and chose not to handle it explicitly. Our distributed checksum mechanism, described later, may detect this type of problem.

[3]For example, if the operating system and the hardware underlying each replica rarely fail and failures at different replicas are independent of each other, it is possible to modify our system so it does not need to flush writes to disk.

In our implementation the master periodically boosts its sequence number by running a full round of the Paxos algorithm, including sending propose messages[4]. Boosting with the right frequency avoids this type of master churn in most cases.

Note that it is possible to extend the concept of leases to all replicas. This will allow any replica with a lease to serve read requests from its local data structure. This extended lease mechanism is useful when read traffic significantly exceeds write traffic. We have examined algorithms for replica leases, but have not implemented them yet.

5.3 Epoch numbers

Requests (by a Chubby client) submitted to a Chubby cell are directed to the current Chubby master replica. From the time when the master replica receives the request to the moment the request causes an update of the underlying database, the replica may have lost its master status. It may even have lost master status and regained it again. Chubby requires an incoming request to be aborted if mastership is lost and/or re-acquired during the handling of the request. We needed a mechanism to reliably detect master turnover and abort operations if necessary.

We solved this problem by introducing a global epoch number with the following semantics. Two requests for the epoch number at the master replica receive the same value iff that replica was master continuously for the time interval between the two requests. The epoch number is stored as an entry in the database, and all database operations are made conditional on the value of the epoch number.

5.4 Group membership

Practical systems must be able to handle changes in the set of replicas. This is referred to as the group membership problem in the literature [3]. Some Paxos papers point out that the Paxos algorithm itself can be used to implement group membership [8]. While group membership with the core Paxos algorithm is straightforward, the exact details are non-trivial when we introduce Multi-Paxos, disk corruptions, etc. Unfortunately the literature does not spell this out, nor does it contain a proof of correctness for algorithms related to group membership changes using Paxos. We had to fill in these gaps to make group membership work in our system. The details – though relatively minor – are subtle and beyond the scope of this paper.

5.5 Snapshots

As described thus far, the repeated application of a consensus algorithm to create a replicated log will lead to an ever growing log. This has two problems: it requires unbounded amounts of disk space; and perhaps worse, it may result in unbounded recovery time since a recovering replica has to replay a potentially long log before it has fully caught up with other replicas. Since the log is typically a sequence of operations to be applied to some data structure, and thus implicitly (through replay) represents a persistent form of that data structure, the problem is to find an alternative persistent representation for the data structure at hand. An obvious mechanism is to persist – or *snapshot* – the data structure directly, at which point the log of operations leading to the current state of the data structure is no longer needed. For example, if the data structure is held in memory, we take a snapshot by serializing it on disk. If the data structure is kept on disk, a snapshot may just be an on-disk copy of it.

By itself, the Paxos framework does not know anything about the data structure we are trying to replicate; its only concern is the consistency of the replicated log. It is the particular application using the Paxos framework that has all the knowledge about the replicated data structure. Thus the application must be responsible for taking snapshots. Our framework provides a mechanism that allows client applications, e.g. our fault-tolerant database, to inform the framework that a snapshot was taken; the client application is free to take a snapshot at any point. When the Paxos framework is informed about a snapshot, it will truncate its log by deleting log entries that precede the snapshot. Should the replica fail, during subsequent recovery

[4]In a loaded system, under one percent of the Paxos instances run the full Paxos algorithm.

it will simply install the latest snapshot and then replay the truncated log to rebuild its state. Snapshots are not synchronized across replicas; each replica independently decides when to create a snapshot.

This mechanism appears straightforward at first and is mentioned briefly in the literature [8]. However, it introduces a fair amount of complexity into the system: the persistent state of a replica now comprises a log *and* a snapshot that have to be maintained consistently. The log is fully under the framework's control, while the snapshot format is application-specific. Some aspects of the snapshot machinery are of particular interest:

- The snapshot and log need to be mutually consistent. Each snapshot needs to have information about its contents relative to the fault-tolerant log. In our framework we introduced the concept of a *snapshot handle* for this purpose. The snapshot handle contains all the Paxos-specific information related to a particular snapshot. When creating a snapshot (which is under control of the application) the corresponding snapshot handle (provided by the framework) needs to be stored by the application as well. When recovering a snapshot, the application must return the snapshot handle to the framework, which in turn will use the information in the handle to coordinate the snapshot with the log.

  Note that the handle is really a snapshot of the Paxos state itself. In our system, it contains the Paxos instance number corresponding to the (log) snapshot and the group membership at that point.

- Taking a snapshot takes time and in some situations we cannot afford to freeze a replica's log while it is taking a snapshot. In our framework, taking a snapshot is split into three phases. First, when the client application decides to take a snapshot, it requests a snapshot handle. Next, the client application takes its snapshot. It may block the system while taking the snapshot, or – more likely – spawn a thread that takes a snapshot while the replica continues to participate in Paxos. The snapshot must correspond to the client state at the log position when the handle was obtained. Thus if the replica continues to participate in Paxos while taking a snapshot, special precautions may have to be taken to snapshot the client's data structure while it is actively updated.[5] Finally, when the snapshot has been taken, the client application informs the framework about the snapshot and passes the corresponding snapshot handle. The framework then truncates the log appropriately.

- Taking a snapshot may fail. Our framework only truncates the log when it is informed that a snapshot has been taken and has received the corresponding snapshot handle. Thus, as long as the client application does not inform the framework, from the framework's viewpoint, no snapshot has been taken. This allows the client application to verify a snapshot's integrity and discard it if necessary. If there is a problem with the snapshot, the client doesn't ask the framework to truncate its log. A client application may even attempt to take several snapshots at the same time using this mechanism.

- While in catch-up, a replica will attempt to obtain missing log records. If it cannot obtain them (because no replica has old-enough log entries readily available), the replica will be told to obtain a snapshot from another replica. This snapshot's handle contains information about the Paxos instance up to which the snapshot captured the state. Once the snapshot has been received and installed, under most circumstances the lagging replica will be close to the leading replica. In order to completely catch-up, the lagging replica asks for and receives the remaining log records from the leading replica to bring it fully up-to-date.

  Note that a leading replica may even create a new snapshot while a lagging replica is installing an older snapshot – in a fault-tolerant system this cannot be avoided. In this scenario, the lagging replica may not be able to obtain any outstanding log records because the snapshot provider (and any other replicas) may have moved ahead in the meantime. The lagging replica will need to obtain a more recent snapshot.

[5] Our first implementation of the fault-tolerant database blocked the system very briefly while making an in-memory copy of the (small) database. It then stored the copied data on disk via a separate thread. Subsequently we implemented virtually pause-less snapshots. We now use a "shadow" data structure to track updates while the underlying database is serialized to disk.

Furthermore, the leading replica may fail after sending its snapshot. The catch-up mechanism must be able to recover from such problems by having the lagging replica contact another leading replica.

- We needed a mechanism to locate recent snapshots. Some applications may choose to transfer snapshots directly between leading and lagging replicas while others may ask a lagging replica to look up a snapshot on GFS. We implemented a general mechanism that allows an application to pass snapshot location information between leading and lagging replicas.

5.6 Database transactions

The database requirements imposed by Chubby are simple: the database needs to store key-value pairs (with keys and values being arbitrary strings), and support common operations such as `insert`, `delete`, `lookup`, an atomic `compare and swap` (`cas`), and iteration over all entries. We implemented a log-structured design using a snapshot of the full database, and a log of database operations to be applied to that snapshot. The log of operations is the Paxos log. The implementation periodically takes a snapshot of the database state and truncates the log accordingly.

The `cas` operation needed to be atomic with respect to other database operations (potentially issued by a different replica). This was easily achieved by submitting all `cas`-related data as a single "value" to Paxos. We realized that we could extend this mechanism to provide transaction-style support without having to implement true database transactions. We describe our solution in more detail because we believe it to be useful in other contexts.

Our implementation hinges around a powerful primitive which we call `MultiOp`. All other database operations except for iteration are implemented as a single call to `MultiOp`. A `MultiOp` is applied atomically and consists of three components:

1. A list of tests called **guard**. Each test in `guard` checks a single entry in the database. It may check for the absence or presence of a value, or compare with a given value. Two different tests in the guard may apply to the same or different entries in the database. All tests in the guard are applied and `MultiOp` returns the results. If all tests are true, `MultiOp` executes `t_op` (see item 2 below), otherwise it executes `f_op` (see item 3 below).

2. A list of database operations called **`t_op`**. Each operation in the list is either an `insert`, `delete`, or `lookup` operation, and applies to a single database entry. Two different operations in the list may apply to the same or different entries in the database. These operations are executed[6] if **guard** evaluates to true.

3. A list of database operations called **`f_op`**. Like `t_op`, but executed if **guard** evaluates to false.

Late in our development (and after we had implemented the database and `MultiOp`), we realized that we also needed epoch numbers to implement database operations for Chubby. With this additional requirement, all Chubby operations became associated with an epoch number and were required to fail if the Paxos epoch number changed. `MultiOp` proved useful in accomodating this new requirement. After we incorporated the Paxos epoch as a database entry, we were able to modify all previous calls to our database to include an additional guard to check for the epoch number.

6 Software Engineering

Fault-tolerant systems are expected to run continuously for long periods of time. Users are much less likely to tolerate bugs than in other systems. For instance, a layout bug exhibited by a document editor may be annoying to a user, but often it is possible to "work around" the issue, even though the bug is really at the

[6] Each `MultiOp` operation is serialized atomically with respect to other operations. The individual operations in the list are executed sequentially on the database.

core of what the software is supposed to do. A bug of similar gravity in a fault-tolerant system may make the system unusable.

We adopted several software engineering methods to give us confidence in the robustness of our implementation. We describe some of the methods we used in this section.

6.1 Expressing the algorithm effectively

Fault-tolerant algorithms are notoriously hard to express correctly, even as pseudo-code. This problem is worse when the code for such an algorithm is intermingled with all the other code that goes into building a complete system. It becomes harder to see the core algorithm, to reason about it, or to debug it when a bug is present. It also makes it difficult to change the core algorithm in response to a requirement change.

We addressed this problem by coding the core algorithm as two explicit state machines. For that purpose, we designed a simple state machine specification language and built a compiler to translate such specifications into C++. The language was designed to be terse so that a full algorithm can be rendered on a single screen. As an additional benefit, the state machine compiler also automatically generates code to log state transitions and measure code coverage to assist in debugging and testing.

We believe that choosing a specification language makes it easier to reason about and modify our state machines than an explicitly coded implementation that is intermingled with the rest of the system. This is illustrated by the following experience. Towards the end of our development of the fault-tolerant log, we had to make a fundamental change in our group membership algorithm. Prior to this change, a replica roughly went through three states. Initially it waited to join the group, then it joined the group, and finally it left the group. Once a replica left the group, it was not allowed to rejoin the group. We felt this approach was best because an intermittently failing replica would not be able to join the group and disrupt it for long. Intermittent failure turned out to be more common than originally anticipated because normal replicas exhibit intermittent failures from time to time. Thus, we needed to change the algorithm to have two states. Either a replica was in the group or it was out. A replica could switch between these two states often during the lifetime of the system. It took us about one hour to make this change and three days to modify our tests accordingly. Had we intermingled our state machines with the rest of the system, this change would have been more difficult to make.

6.2 Runtime consistency checking

The chance for inconsistencies increases with the size of the code base, the duration of a project, and the number of people working simultaneously on the same code. We used various active self-checking mechanisms such as the liberal use of `assert` statements, and explicit verification code that tests data structures for consistency.

For example, we used the following database consistency check. The master periodically submits a `checksum` request to the database log. On receipt of this request, each replica computes a checksum of its local database[7]. Since the Paxos log serializes all operations identically on all replicas, we expect all replicas to compute the same checksum. After the master completes a checksum computation, it sends its checksum to all replicas which compare the master's checksum with their computed checksum.

We have had three database inconsistency incidents thus far:

- The first incident was due to an operator error.
- We have not found an explanation for the second incident. On replaying the faulty replica's log we found that it was consistent with the other replicas. Thus it is possible that this problem was caused by a random hardware memory corruption.
- We suspect the third was due to an illegal memory access from errant code in the included codebase (which is of considerable size). To protect against this possibility in the future, we maintain a second database of checksums and double-check every database access against the database of checksums.

[7] We use a shadow datastructure to handle database operations concurrently with the checksum computation.

In all three cases manual intervention appeared to resolve the problem before it became visible to Chubby.

6.3 Testing

Given the current state of the art, it is unrealistic to prove a real system such as ours correct. To achieve robustness, the best practical solution in addition to meticulous software engineering is to test a system thoroughly. Our system was designed to be testable from the onset and now contains an extensive suite of tests. In this section we describe two tests that take the system through a long sequence of random failures and verify that it behaves as expected. Both tests can run in one of two modes:

1. **Safety mode**. In this mode, the test verifies that the system is consistent. However, the system is not required to make any progress. For example, it is acceptable for an operation to fail to complete or to report that the system is unavailable.

2. **Liveness mode**. In this mode, the test verifies that the system is consistent and is making progress. All operations are expected to complete and the system is required to be consistent.

Our tests start in safety mode and inject random failures into the system. After running for a predetermined period of time, we stop injecting failures and give the system time to fully recover. Then we switch the test to liveness mode. The purpose for the liveness test is to verify that the system does not deadlock after a sequence of failures.

One of our tests verifies the fault-tolerant log. It simulates a distributed system consisting of a random number of replicas and takes our fault-tolerant log through a random sequence of network outages, message delays, timeouts, process crashes and recoveries, file corruptions, schedule interleavings, etc. We wanted this test to be repeatable to aid in debugging. To this end, we use a random number generator to determine the schedule of failures. The seed for the random number generator is given at the beginning of the test run. We ensure that two test runs with the same random number seed are identical by running the test in a single thread to remove unwanted non-determinism from multi-threading. This is possible because the fault-tolerant log does not create its own threads and can run in a single-threaded environment (even though it normally runs in a multi-threaded environment).

Each test execution reports success or failure. If a test fails, we rerun that test with the failing random number seed and with detailed logging turned on in a debugger to determine what went wrong. This is possible because these tests are repeatable.

This test proved useful in finding various subtle protocol errors, including errors in our group membership implementation, and our modifications to deal with corrupted disks. In order to measure the strength of this test, we left some protocol bugs found during code and design reviews in the system, and verified that our test system detected these bugs. After a number of bug fixes, the test became very stable. In order to improve its bug yield, we started running this test on a farm of several hundred Google machines at a time. We found additional bugs, some of which took weeks of simulated execution time (at extremely high failure rates) to find.

Another test verifies robustness of the new Chubby system against lower-level system and hardware failures. We implemented several hooks in our fault-tolerant log to inject failures. The test randomly invokes these hooks and verifies that higher levels of the system can cope. Our hooks can be used to crash a replica, disconnect it from other replicas for a period of time or force a replica to pretend that it is no longer the master. This test found five subtle bugs in Chubby related to master failover in its first two weeks. In the same vein, we built a filesystem with hooks to programmatically inject failures and are using it to test our ability to deal with filesystem failures.

In closing we point out a challenge that we faced in testing our system for which we have no systematic solution. By their very nature, fault-tolerant systems try to mask problems. Thus they can mask bugs or configuration problems while insidiously lowering their own fault-tolerance. For example, we have observed the following scenario. We once started a system with five replicas, but misspelled the name of one of the replicas in the initial group. The system appeared to run correctly as the four correctly configured replicas

were able to make progress. Further, the fifth replica continunously ran in catch-up mode[8] and therefore appeared to run correctly as well. However in this configuration the system only tolerates one faulty replica instead of the expected two. We now have processes in place to detect this particular type of problem. We have no way of knowing if there are other bugs/misconfigurations that are masked by fault-tolerance.

6.4 Concurrency

At the onset of the project we were concerned about the problem of testing concurrent fault-tolerant code. In particular, we wanted our tests to be repeatable. As described earlier, our fault-tolerant log doesn't contain any of its own threads (even though it can handle concurrent requests on different threads). Threading is introduced at the edges of the code – where we receive calls from the networking layer. By making our tests repeatable, we were able to hunt down several obscure protocol errors during testing.

As the project progressed, we had to make several subsystems more concurrent than we had intended and sacrifice repeatability. Chubby is multi-threaded at its core, thus we cannot run repeatable tests against the complete system. Next we had to make our database multi-threaded so it could take snapshots, compute checksums and process iterators while concurrently serving database requests. Finally, we were forced to make the code that handles the local copy of the log multi-threaded as well (the exact reason why is beyond the scope of this paper).

In summary, we believe that we set ourselves the right goals for repeatability of executions by constraining concurrency. Unfortunately, as the product needs grew we were unable to adhere to these goals.

7 Unexpected failures

So far, our system has logged well over 100 machine years of execution in production. In this period we have witnessed the following unexpected failure scenarios:

- Our first release shipped with ten times the number of worker threads as the original Chubby system. We hoped this change would enable us to handle more requests. Unfortunately, under load, the worker threads ended up starving some other key threads and caused our system to time out frequently. This resulted in rapid master failover, followed by en-masse migrations of large numbers of clients to the new master which caused the new master to be overwhelmed, followed by additional master failovers, and so on.

  When this problem first appeared, the precise cause was unknown and we had to protect ourselves from a potentially dangerous bug in our system. We decided to err on the side of caution and to rollback our system to the old version of Chubby (based on 3DB) in one of our data centers. At that point, the rollback mechanism was not properly documented (because we never expected to use it), its use was non-intuitive, the operator performing the roll-back had no experience with it, and when the rollback was performed, no member of the development team was present. As a result, an old snapshot was accidentally used for the rollback. By the time we discovered the error, we had lost 15 hours of data and several key datasets had to be rebuilt.

- When we tried to upgrade this Chubby cell again a few months later, our upgrade script failed because we had omitted to delete files generated by the failed upgrade from the past. The cell ended up running with a months-old snapshot for a few minutes before we discovered the problem. This caused us to lose about 30 minutes of data. Fortunately all of Chubby's clients recovered from this outage.

- A few months after our initial release, we realized that the semantics provided by our database were different from what Chubby expected. If Chubby submitted an operation to the database, and the database lost its master status, Chubby expected the operation to fail. With our system, a replica

[8] In our implementation, a replica that is not (yet) a group member runs in catch-up mode to stay up-to-date. This allows us to keep a future group member "warm" so it can become an active member immediately after joining the group.

could be re-installed as master during the database operation and the operation could succeed. The fix required a substantial rework of the integration layer between Chubby and our framework (we needed to implement epoch numbers). `MultiOp` proved to be helpful in solving this unexpected problem – an indication that `MultiOp` is a powerful primitive.

- As mentioned before, on three occasions we discovered that one of the database replicas was different from the others in that Chubby cell. We found this problem because our system periodically takes checksums of all replicas and then compares them.
- Our upgrade script which is responsible for migrating cells from the 3DB version of Chubby to the Paxos version has failed several times for a variety of reasons. For example, it once failed because a basic Google program was not installed on one of our cells.
- We have encountered failures due to bugs in the underlying operating system. For example in our version of the Linux 2.4 kernel, when we try to flush a small file to disk, the call can hang for a long time if there are a lot of buffered writes to other files. This happens immediately after we write a database snapshot to disk. In this case, we observed that it could take several seconds for the kernel to flush an unrelated small write to the Paxos log. Our workaround is to write *all* large files in small chunks, with a flush to disk after each small chunk. While this hurts the performance of the write slightly, it protects the more critical log writes from unexpected delays.

A small number of failures in 100 machine years would be considered excellent behavior for most production systems. However, we consider the current failure rate too high for Chubby and we have determined that we need to reduce it further.

Three of the failures occurred during upgrade (or rollback). Every time we encountered a problem during upgrade, we updated our upgrade script accordingly. Once a cell is upgraded, this type of failure will disappear.

Two of the failures were from bugs that have since been fixed. To reduce the probability of other bugs, we continue to improve and run the Chubby verification test outlined earlier.

Two of our unexpected problems relate to operator error during rollout of a new release and caused us to lose data. At Google, the day-to-day monitoring and management of our systems is done by system operators. While they are very competent, they are usually not part of the development team that built the system, and therefore not familiar with its intricate details. This may lead to the occasional operator error in unforseen situations. We now rely on carefully written and well-tested scripts to automate rollout and minimize operator involvement. As a result our most recent major release of Chubby was rolled out across hundreds of machines without incident, while serving life traffic.

One of the failures was due to memory corruption. Because our system is log-structured and maintains several days of log data and snapshots, it was possible to replay the database upto the exact point at which the problem appears. We were able to verify that our logs were correct and conclude that the memory corruption occurred from errant software or due to hardware problems. We added additional checksum data to detect this type of problem in the future and will crash a replica when it detects this problem.

8 Measurements

The initial goal of our system was to replace 3DB with our own database. Thus our system had to demonstrate equal or superior performance relative to 3DB. We measured the performance of a complete Chubby system (clients, server, including network latency) using our fault-tolerant replicated database. We also benchmarked this system against an identical system based on 3DB (see Table 1). For our tests, we ran two copies of Chubby on the same set of 5 servers (typical Pentium®-class machines). One copy of Chubby used our database while the other copy used 3DB. We ran Chubby clients on workstations to generate load on the servers. For our tests, we measured total system throughput. Each call includes the Chubby client, the network, the Chubby server and our fault-tolerant database. While this test underestimates the performance of our database, it gives a sense of the full system throughput of a system based on Paxos.

Test	# workers	file size (bytes)	Paxos-Chubby (100MB DB)	3DB-Chubby (small database)	Comparison
Ops/s Throughput	1	5	91 ops/sec	75 ops/sec	1.2x
Ops/s Throughput	10	5	490 ops/sec	134 ops/sec	3.7x
Ops/s Throughput	20	5	640 ops/sec	178 ops/sec	3.6x
MB/s Throughput	1	8 KB	345 KB/s	172 KB/s	2x
MB/s Throughput	4	8 KB	777 - 949 KB/s	217 KB/s	3.6 - 4.4x
MB/s Throughput	1	32 KB	672 - 822 KB/s	338 KB/s	2.0 - 2.4x

Table 1: Comparing our system with 3DB (higher numbers are better).

Even though read requests to Chubby dominate in practice, we designed our tests to be write intensive. This is because read requests are completely handled on the master, which typically has a lease, and do not exercise the Paxos algorithm.

In our test, each worker repeatedly creates a file in Chubby and waits for Chubby to return before creating the file again. Thus each operation makes one write call to the underlying database. If the contents of the file are small and there is a single worker, the test measures the latency of the system. If the contents of the file are large, the test measures the throughput of the system in MB/s. By using multiple concurrent workers, we were also able to measure the throughput of the system in submissions/s.

All tests with more than one worker show the effect of batching a collection of submitted values. It should be possible to achieve some speedup with 3DB by bundling a collection of updates in a database transaction. The last two throughput tests show the effect of taking snapshots. This system was configured to take a snapshot whenever the replicated log size exceeded 100 MB. In these two tests, the system takes snapshots roughly every 100 seconds. When taking a snapshot, the system makes another copy of the database and writes it to disk. As a result, its performance temporarily drops off.

Our system is by no means optimized for performance, and we believe that there is a lot of room to make it faster. However, given the performance improvement over 3DB, further optimizations are not a priority at this time.

9 Summary and open problems

We have described our implementation of a fault-tolerant database, based on the Paxos consensus algorithm. Despite the large body of literature in the field, algorithms dating back more then 15 years, and experience of our team (one of us has designed a similar system before and the others have built other types of complex systems in the past), it was significantly harder to build this system then originally anticipated. We attribute this to several shortcomings in the field:

- There are significant gaps between the description of the Paxos algorithm and the needs of a real-world system. In order to build a real-world system, an expert needs to use numerous ideas scattered in the literature and make several relatively small protocol extensions. The cumulative effort will be substantial and the final system will be based on an unproven protocol.
- The fault-tolerance computing community has not developed the tools to make it easy to implement their algorithms.
- The fault-tolerance computing community has not paid enough attention to testing, a key ingredient for building fault-tolerant systems.

As a result, the core algorithms work remains relatively theoretical and is not as accessible to a larger computing community as it could be. We believe that in order to make a greater impact, researchers in the field should focus on addressing these shortcomings.

In contrast, consider the field of compiler construction. Though concepts in that field are complex, they have been made accessible to a wide audience. Industrial-strength parsing tools such `yacc` [6] appeared not too long after the theory of parsing was well-understood. Not only are there now many front-end tools such as `ANTLR` [15] or `CoCo/R` [13]; but there are also tree-rewriting tools helping with optimizations and instruction selection, assemblers helping with binary code generation, and so forth. Thus, in this area of software engineering, an entire family of tools has emerged, making the construction of a compiler significantly easier or at least less error-prone. Disciplines within the field of compiler construction, such as parsing, which were once at the cutting edge of research, are now considered "solved" and are routinely taught at the undergraduate level in many schools.

It appears that the fault-tolerant distributed computing community has not developed the tools and know-how to close the gaps between theory and practice with the same vigor as for instance the compiler community. Our experience suggests that these gaps are non-trivial and that they merit attention by the research community.

10 Acknowledgments

Many people at Google helped us with this project. Mike Burrows who implemented Chubby suggested that we replace 3DB with a Paxos-based system. He and Sharon Perl reviewed our designs and provided excellent feedback. They introduced us to the mechanism for handling disk corruptions and suggested that we implement master leases. Michal Cierniak ported the original state machine compiler from Perl to C++ and made substantial modifications (it is now being used elsewhere at Google as well). Vadim Furman helped us write the Chubby verification test. Salim Virji and his team were responsible for the roll-out of our system across Google data centers.

Mike Burrows, Bill Coughran, Gregory Eitzman, Peter Mckenzie, Sharon Perl, Rob Pike, David Presotto, Sean Quinlan, and Salim Virji reviewed earlier versions of this paper and provided valuable feedback.

References

[1] Burrows, M. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, pp. 335-350

[2] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E. Bigtable: A distributed storage system for structured data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, pp. 205-218

[3] Cristian, F. Reaching agreement on processor-group membership in synchronous distributed systems. *Distributed Computing 4*, 4 (1991), 175–188.

[4] Ghemawat, S., Gobioff, H., and Leung, S.-T. The Google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles* (Dec. 2003), pp. 29–43.

[5] Gray, C., Cheriton, D. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles* (1989), pp. 202–210.

[6] Johnson, S. C. Yacc: Yet another compiler-compiler.

[7] Lamport, Shostak, and Pease. The byzantine generals problem. In *Advances in Ultra-Dependable Distributed Systems, N. Suri, C. J. Walter, and M. M. Hugue (Eds.), IEEE Computer Society Press.* 1995.

[8] Lamport, L. The part-time parliament. *ACM Transactions on Computer Systems 16*, 2 (1998), 133–169.

[9] Lamport, L. Paxos made simple. *ACM SIGACT News 32*, 4 (Dec. 2001), 18–25.

[10] Lampson, B. W. How to build a highly available system using consensus. In *10th International Workshop on Distributed Algorithms (WDAG 96)* (1996), Babaoglu and Marzullo, Eds., vol. 1151, Springer-Verlag, Berlin Germany, pp. 1–17.

[11] Lee, E. K., and Thekkath, C. A. Petal: Distributed virtual disks. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems* (Cambridge, MA, 1996), pp. 84–92.

[12] MacCormick, J., Murphy, N., Najork, M., Thekkath, C. A., and Zhou, L. Boxwood: Abstractions as the foundation for storage infrastructure. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation* (2004), pp. 105–120.

[13] Moessenboeck, H. A generator for production quality compilers. In *Proceedings of the 3rd International Workshop on Compiler Compilers - Lecture Notes in Computer Science 477* (Berlin, Heidelberg, New York, Tokyo, 1990), Springer-Verlag, pp. 42–55.

[14] Oki, Brian M., and Liskov, Barbara H. Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems. In *Proceedings of the 7th annual ACM Symposium on Principles of Distributed Computing* (1988), pp. 8–17.

[15] Parr, T. J., and QUONG, R. W. Antlr: A predicated-ll(k) parser generator. *Software–Practice and Experience 25*, 7 (JULY 1995), 789–810.

[16] Prisco, R. D., Lampson, B. W., and Lynch, N. A. Revisiting the paxos algorithm. In *11th International Workshop on Distributed Algorithms (WDAG 96)* (1997), pp. 111–125.

[17] Schneider, F. B. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys 22*, 4 (1990), 299–319.

[18] von Neumann, J. Probabilistic logics and synthesis of reliable organisms from unreliable components. *Automata Studies* (1956), 43–98.

Chapter 16

Spanner: Google's Globally-Distributed Database

Authors:
James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, Dale Woodford

Publication year: 2012

Spanner: Google's Globally-Distributed Database

James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, Dale Woodford

Google, Inc.

Abstract

Spanner is Google's scalable, multi-version, globally-distributed, and synchronously-replicated database. It is the first system to distribute data at global scale and support externally-consistent distributed transactions. This paper describes how Spanner is structured, its feature set, the rationale underlying various design decisions, and a novel time API that exposes clock uncertainty. This API and its implementation are critical to supporting external consistency and a variety of powerful features: non-blocking reads in the past, lock-free read-only transactions, and atomic schema changes, across all of Spanner.

1 Introduction

Spanner is a scalable, globally-distributed database designed, built, and deployed at Google. At the highest level of abstraction, it is a database that shards data across many sets of Paxos [21] state machines in datacenters spread all over the world. Replication is used for global availability and geographic locality; clients automatically failover between replicas. Spanner automatically reshards data across machines as the amount of data or the number of servers changes, and it automatically migrates data across machines (even across datacenters) to balance load and in response to failures. Spanner is designed to scale up to millions of machines across hundreds of datacenters and trillions of database rows.

Applications can use Spanner for high availability, even in the face of wide-area natural disasters, by replicating their data within or even across continents. Our initial customer was F1 [35], a rewrite of Google's advertising backend. F1 uses five replicas spread across the United States. Most other applications will probably replicate their data across 3 to 5 datacenters in one geographic region, but with relatively independent failure modes. That is, most applications will choose lower latency over higher availability, as long as they can survive 1 or 2 datacenter failures.

Spanner's main focus is managing cross-datacenter replicated data, but we have also spent a great deal of time in designing and implementing important database features on top of our distributed-systems infrastructure. Even though many projects happily use Bigtable [9], we have also consistently received complaints from users that Bigtable can be difficult to use for some kinds of applications: those that have complex, evolving schemas, or those that want strong consistency in the presence of wide-area replication. (Similar claims have been made by other authors [37].) Many applications at Google have chosen to use Megastore [5] because of its semi-relational data model and support for synchronous replication, despite its relatively poor write throughput. As a consequence, Spanner has evolved from a Bigtable-like versioned key-value store into a temporal multi-version database. Data is stored in schematized semi-relational tables; data is versioned, and each version is automatically timestamped with its commit time; old versions of data are subject to configurable garbage-collection policies; and applications can read data at old timestamps. Spanner supports general-purpose transactions, and provides a SQL-based query language.

As a globally-distributed database, Spanner provides several interesting features. First, the replication configurations for data can be dynamically controlled at a fine grain by applications. Applications can specify constraints to control which datacenters contain which data, how far data is from its users (to control read latency), how far replicas are from each other (to control write latency), and how many replicas are maintained (to control durability, availability, and read performance). Data can also be dynamically and transparently moved between datacenters by the system to balance resource usage across datacenters. Second, Spanner has two features that are difficult to implement in a distributed database: it

provides externally consistent [16] reads and writes, and globally-consistent reads across the database at a timestamp. These features enable Spanner to support consistent backups, consistent MapReduce executions [12], and atomic schema updates, all at global scale, and even in the presence of ongoing transactions.

These features are enabled by the fact that Spanner assigns globally-meaningful commit timestamps to transactions, even though transactions may be distributed. The timestamps reflect serialization order. In addition, the serialization order satisfies external consistency (or equivalently, linearizability [20]): if a transaction T_1 commits before another transaction T_2 starts, then T_1's commit timestamp is smaller than T_2's. Spanner is the first system to provide such guarantees at global scale.

The key enabler of these properties is a new TrueTime API and its implementation. The API directly exposes clock uncertainty, and the guarantees on Spanner's timestamps depend on the bounds that the implementation provides. If the uncertainty is large, Spanner slows down to wait out that uncertainty. Google's cluster-management software provides an implementation of the TrueTime API. This implementation keeps uncertainty small (generally less than 10ms) by using multiple modern clock references (GPS and atomic clocks).

Section 2 describes the structure of Spanner's implementation, its feature set, and the engineering decisions that went into their design. Section 3 describes our new TrueTime API and sketches its implementation. Section 4 describes how Spanner uses TrueTime to implement externally-consistent distributed transactions, lock-free read-only transactions, and atomic schema updates. Section 5 provides some benchmarks on Spanner's performance and TrueTime behavior, and discusses the experiences of F1. Sections 6, 7, and 8 describe related and future work, and summarize our conclusions.

2 Implementation

This section describes the structure of and rationale underlying Spanner's implementation. It then describes the *directory* abstraction, which is used to manage replication and locality, and is the unit of data movement. Finally, it describes our data model, why Spanner looks like a relational database instead of a key-value store, and how applications can control data locality.

A Spanner deployment is called a *universe*. Given that Spanner manages data globally, there will be only a handful of running universes. We currently run a test/playground universe, a development/production universe, and a production-only universe.

Spanner is organized as a set of *zones*, where each zone is the rough analog of a deployment of Bigtable servers [9]. Zones are the unit of administrative deployment. The set of zones is also the set of locations across which data can be replicated. Zones can be added to or removed from a running system as new datacenters are brought into service and old ones are turned off, respectively. Zones are also the unit of physical isolation: there may be one or more zones in a datacenter, for example, if different applications' data must be partitioned across different sets of servers in the same datacenter.

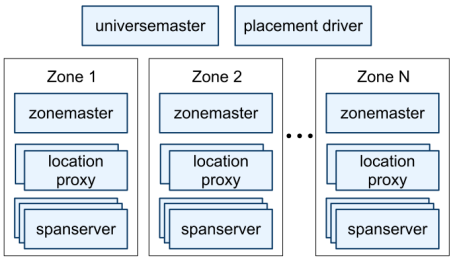


Figure 1: Spanner server organization.

Figure 1 illustrates the servers in a Spanner universe. A zone has one *zonemaster* and between one hundred and several thousand *spanservers*. The former assigns data to spanservers; the latter serve data to clients. The per-zone *location proxies* are used by clients to locate the spanservers assigned to serve their data. The *universe master* and the *placement driver* are currently singletons. The universe master is primarily a console that displays status information about all the zones for interactive debugging. The placement driver handles automated movement of data across zones on the timescale of minutes. The placement driver periodically communicates with the spanservers to find data that needs to be moved, either to meet updated replication constraints or to balance load. For space reasons, we will only describe the spanserver in any detail.

2.1 Spanserver Software Stack

This section focuses on the spanserver implementation to illustrate how replication and distributed transactions have been layered onto our Bigtable-based implementation. The software stack is shown in Figure 2. At the bottom, each spanserver is responsible for between 100 and 1000 instances of a data structure called a *tablet*. A tablet is similar to Bigtable's tablet abstraction, in that it implements a bag of the following mappings:

$$(\text{key:string}, \text{timestamp:int64}) \rightarrow \text{string}$$

Unlike Bigtable, Spanner assigns timestamps to data, which is an important way in which Spanner is more like a multi-version database than a key-value store. A

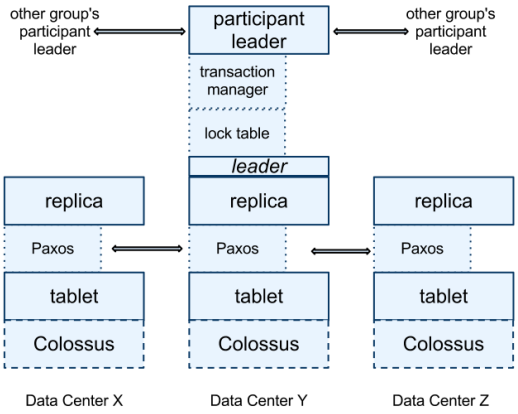


Figure 2: Spanserver software stack.

tablet's state is stored in set of B-tree-like files and a write-ahead log, all on a distributed file system called Colossus (the successor to the Google File System [15]).

To support replication, each spanserver implements a single Paxos state machine on top of each tablet. (An early Spanner incarnation supported multiple Paxos state machines per tablet, which allowed for more flexible replication configurations. The complexity of that design led us to abandon it.) Each state machine stores its metadata and log in its corresponding tablet. Our Paxos implementation supports long-lived leaders with time-based leader leases, whose length defaults to 10 seconds. The current Spanner implementation logs every Paxos write twice: once in the tablet's log, and once in the Paxos log. This choice was made out of expediency, and we are likely to remedy this eventually. Our implementation of Paxos is pipelined, so as to improve Spanner's throughput in the presence of WAN latencies; but writes are applied by Paxos in order (a fact on which we will depend in Section 4).

The Paxos state machines are used to implement a consistently replicated bag of mappings. The key-value mapping state of each replica is stored in its corresponding tablet. Writes must initiate the Paxos protocol at the leader; reads access state directly from the underlying tablet at any replica that is sufficiently up-to-date. The set of replicas is collectively a *Paxos group*.

At every replica that is a leader, each spanserver implements a *lock table* to implement concurrency control. The lock table contains the state for two-phase locking: it maps ranges of keys to lock states. (Note that having a long-lived Paxos leader is critical to efficiently managing the lock table.) In both Bigtable and Spanner, we designed for long-lived transactions (for example, for report generation, which might take on the order of minutes), which perform poorly under optimistic concurrency control in the presence of conflicts. Operations

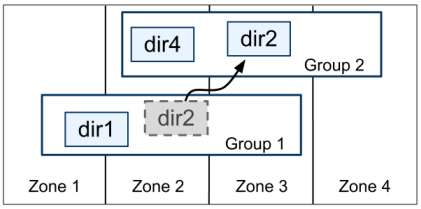


Figure 3: Directories are the unit of data movement between Paxos groups.

that require synchronization, such as transactional reads, acquire locks in the lock table; other operations bypass the lock table.

At every replica that is a leader, each spanserver also implements a *transaction manager* to support distributed transactions. The transaction manager is used to implement a *participant leader*; the other replicas in the group will be referred to as *participant slaves*. If a transaction involves only one Paxos group (as is the case for most transactions), it can bypass the transaction manager, since the lock table and Paxos together provide transactionality. If a transaction involves more than one Paxos group, those groups' leaders coordinate to perform two-phase commit. One of the participant groups is chosen as the coordinator: the participant leader of that group will be referred to as the *coordinator leader*, and the slaves of that group as *coordinator slaves*. The state of each transaction manager is stored in the underlying Paxos group (and therefore is replicated).

2.2 Directories and Placement

On top of the bag of key-value mappings, the Spanner implementation supports a bucketing abstraction called a *directory*, which is a set of contiguous keys that share a common prefix. (The choice of the term *directory* is a historical accident; a better term might be *bucket*.) We will explain the source of that prefix in Section 2.3. Supporting directories allows applications to control the locality of their data by choosing keys carefully.

A directory is the unit of data placement. All data in a directory has the same replication configuration. When data is moved between Paxos groups, it is moved directory by directory, as shown in Figure 3. Spanner might move a directory to shed load from a Paxos group; to put directories that are frequently accessed together into the same group; or to move a directory into a group that is closer to its accessors. Directories can be moved while client operations are ongoing. One could expect that a 50MB directory can be moved in a few seconds.

The fact that a Paxos group may contain multiple directories implies that a Spanner tablet is different from

a Bigtable tablet: the former is not necessarily a single lexicographically contiguous partition of the row space. Instead, a Spanner tablet is a container that may encapsulate multiple partitions of the row space. We made this decision so that it would be possible to colocate multiple directories that are frequently accessed together.

Movedir is the background task used to move directories between Paxos groups [14]. Movedir is also used to add or remove replicas to Paxos groups [25], because Spanner does not yet support in-Paxos configuration changes. Movedir is not implemented as a single transaction, so as to avoid blocking ongoing reads and writes on a bulky data move. Instead, movedir registers the fact that it is starting to move data and moves the data in the background. When it has moved all but a nominal amount of the data, it uses a transaction to atomically move that nominal amount and update the metadata for the two Paxos groups.

A directory is also the smallest unit whose geographic-replication properties (or *placement*, for short) can be specified by an application. The design of our placement-specification language separates responsibilities for managing replication configurations. Administrators control two dimensions: the number and types of replicas, and the geographic placement of those replicas. They create a menu of named options in these two dimensions (e.g., *North America, replicated 5 ways with 1 witness*). An application controls how data is replicated, by tagging each database and/or individual directories with a combination of those options. For example, an application might store each end-user's data in its own directory, which would enable user A's data to have three replicas in Europe, and user B's data to have five replicas in North America.

For expository clarity we have over-simplified. In fact, Spanner will shard a directory into multiple *fragments* if it grows too large. Fragments may be served from different Paxos groups (and therefore different servers). Movedir actually moves fragments, and not whole directories, between groups.

2.3 Data Model

Spanner exposes the following set of data features to applications: a data model based on schematized semi-relational tables, a query language, and general-purpose transactions. The move towards supporting these features was driven by many factors. The need to support schematized semi-relational tables and synchronous replication is supported by the popularity of Megastore [5]. At least 300 applications within Google use Megastore (despite its relatively low performance) because its data model is simpler to manage than Bigtable's, and because of its support for synchronous replication across datacenters. (Bigtable only supports eventually-consistent replication across datacenters.) Examples of well-known Google applications that use Megastore are Gmail, Picasa, Calendar, Android Market, and AppEngine. The need to support a SQL-like query language in Spanner was also clear, given the popularity of Dremel [28] as an interactive data-analysis tool. Finally, the lack of cross-row transactions in Bigtable led to frequent complaints; Percolator [32] was in part built to address this failing. Some authors have claimed that general two-phase commit is too expensive to support, because of the performance or availability problems that it brings [9, 10, 19]. We believe it is better to have application programmers deal with performance problems due to overuse of transactions as bottlenecks arise, rather than always coding around the lack of transactions. Running two-phase commit over Paxos mitigates the availability problems.

The application data model is layered on top of the directory-bucketed key-value mappings supported by the implementation. An application creates one or more *databases* in a universe. Each database can contain an unlimited number of schematized *tables*. Tables look like relational-database tables, with rows, columns, and versioned values. We will not go into detail about the query language for Spanner. It looks like SQL with some extensions to support protocol-buffer-valued fields.

Spanner's data model is not purely relational, in that rows must have names. More precisely, every table is required to have an ordered set of one or more primary-key columns. This requirement is where Spanner still looks like a key-value store: the primary keys form the name for a row, and each table defines a mapping from the primary-key columns to the non-primary-key columns. A row has existence only if some value (even if it is NULL) is defined for the row's keys. Imposing this structure is useful because it lets applications control data locality through their choices of keys.

Figure 4 contains an example Spanner schema for storing photo metadata on a per-user, per-album basis. The schema language is similar to Megastore's, with the additional requirement that every Spanner database must be partitioned by clients into one or more hierarchies of tables. Client applications declare the hierarchies in database schemas via the `INTERLEAVE IN` declarations. The table at the top of a hierarchy is a *directory table*. Each row in a directory table with key K, together with all of the rows in descendant tables that start with K in lexicographic order, forms a directory. `ON DELETE CASCADE` says that deleting a row in the directory table deletes any associated child rows. The figure also illustrates the interleaved layout for the example database: for

```
CREATE TABLE Users {
  uid INT64 NOT NULL, email STRING
} PRIMARY KEY (uid), DIRECTORY;

CREATE TABLE Albums {
  uid INT64 NOT NULL, aid INT64 NOT NULL,
  name STRING
} PRIMARY KEY (uid, aid),
  INTERLEAVE IN PARENT Users ON DELETE CASCADE;
```

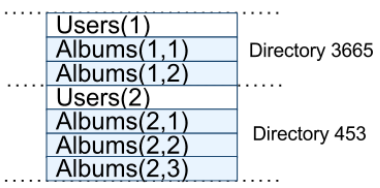


Figure 4: Example Spanner schema for photo metadata, and the interleaving implied by `INTERLEAVE IN`.

example, `Albums(2,1)` represents the row from the `Albums` table for `user_id 2, album_id 1`. This interleaving of tables to form directories is significant because it allows clients to describe the locality relationships that exist between multiple tables, which is necessary for good performance in a sharded, distributed database. Without it, Spanner would not know the most important locality relationships.

3 TrueTime

Method	Returns
TT.now()	*TTinterval*: [*earliest*, *latest*]
TT.after(t)	true if t has definitely passed
TT.before(t)	true if t has definitely not arrived

Table 1: TrueTime API. The argument t is of type *TTstamp*.

This section describes the TrueTime API and sketches its implementation. We leave most of the details for another paper: our goal is to demonstrate the power of having such an API. Table 1 lists the methods of the API. TrueTime explicitly represents time as a *TTinterval*, which is an interval with bounded time uncertainty (unlike standard time interfaces that give clients no notion of uncertainty). The endpoints of a *TTinterval* are of type *TTstamp*. The *TT.now*() method returns a *TTinterval* that is guaranteed to contain the absolute time during which *TT.now*() was invoked. The time epoch is analogous to UNIX time with leap-second smearing. Define the instantaneous error bound as ϵ, which is half of the interval's width, and the average error bound as $\bar{\epsilon}$. The *TT.after*() and *TT.before*() methods are convenience wrappers around *TT.now*().

Denote the absolute time of an event e by the function $t_{abs}(e)$. In more formal terms, TrueTime guarantees that for an invocation $tt = TT.now()$, $tt.earliest \leq t_{abs}(e_{now}) \leq tt.latest$, where e_{now} is the invocation event.

The underlying time references used by TrueTime are GPS and atomic clocks. TrueTime uses two forms of time reference because they have different failure modes. GPS reference-source vulnerabilities include antenna and receiver failures, local radio interference, correlated failures (e.g., design faults such as incorrect leap-second handling and spoofing), and GPS system outages. Atomic clocks can fail in ways uncorrelated to GPS and each other, and over long periods of time can drift significantly due to frequency error.

TrueTime is implemented by a set of *time master* machines per datacenter and a *timeslave daemon* per machine. The majority of masters have GPS receivers with dedicated antennas; these masters are separated physically to reduce the effects of antenna failures, radio interference, and spoofing. The remaining masters (which we refer to as *Armageddon masters*) are equipped with atomic clocks. An atomic clock is not that expensive: the cost of an Armageddon master is of the same order as that of a GPS master. All masters' time references are regularly compared against each other. Each master also cross-checks the rate at which its reference advances time against its own local clock, and evicts itself if there is substantial divergence. Between synchronizations, Armageddon masters advertise a slowly increasing time uncertainty that is derived from conservatively applied worst-case clock drift. GPS masters advertise uncertainty that is typically close to zero.

Every daemon polls a variety of masters [29] to reduce vulnerability to errors from any one master. Some are GPS masters chosen from nearby datacenters; the rest are GPS masters from farther datacenters, as well as some Armageddon masters. Daemons apply a variant of Marzullo's algorithm [27] to detect and reject liars, and synchronize the local machine clocks to the non-liars. To protect against broken local clocks, machines that exhibit frequency excursions larger than the worst-case bound derived from component specifications and operating environment are evicted.

Between synchronizations, a daemon advertises a slowly increasing time uncertainty. ϵ is derived from conservatively applied worst-case local clock drift. ϵ also depends on time-master uncertainty and communication delay to the time masters. In our production environment, ϵ is typically a sawtooth function of time, varying from about 1 to 7 ms over each poll interval. $\bar{\epsilon}$ is therefore 4 ms most of the time. The daemon's poll interval is currently 30 seconds, and the current applied drift rate is set at 200 microseconds/second, which together account

Operation	Timestamp Discussion	Concurrency Control	Replica Required
Read-Write Transaction	§ 4.1.2	pessimistic	leader
Read-Only Transaction	§ 4.1.4	lock-free	leader for timestamp; any for read, subject to § 4.1.3
Snapshot Read, client-provided timestamp	—	lock-free	any, subject to § 4.1.3
Snapshot Read, client-provided bound	§ 4.1.3	lock-free	any, subject to § 4.1.3

Table 2: Types of reads and writes in Spanner, and how they compare.

for the sawtooth bounds from 0 to 6 ms. The remaining 1 ms comes from the communication delay to the time masters. Excursions from this sawtooth are possible in the presence of failures. For example, occasional time-master unavailability can cause datacenter-wide increases in ϵ. Similarly, overloaded machines and network links can result in occasional localized ϵ spikes.

4 Concurrency Control

This section describes how TrueTime is used to guarantee the correctness properties around concurrency control, and how those properties are used to implement features such as externally consistent transactions, lock-free read-only transactions, and non-blocking reads in the past. These features enable, for example, the guarantee that a whole-database audit read at a timestamp t will see exactly the effects of every transaction that has committed as of t.

Going forward, it will be important to distinguish writes as seen by Paxos (which we will refer to as *Paxos writes* unless the context is clear) from Spanner client writes. For example, two-phase commit generates a Paxos write for the prepare phase that has no corresponding Spanner client write.

4.1 Timestamp Management

Table 2 lists the types of operations that Spanner supports. The Spanner implementation supports *read-write transactions*, *read-only transactions* (predeclared snapshot-isolation transactions), and *snapshot reads*. Standalone writes are implemented as read-write transactions; non-snapshot standalone reads are implemented as read-only transactions. Both are internally retried (clients need not write their own retry loops).

A read-only transaction is a kind of transaction that has the performance benefits of snapshot isolation [6]. A read-only transaction must be predeclared as not having any writes; it is not simply a read-write transaction without any writes. Reads in a read-only transaction execute at a system-chosen timestamp without locking, so that incoming writes are not blocked. The execution of the reads in a read-only transaction can proceed on any replica that is sufficiently up-to-date (Section 4.1.3).

A snapshot read is a read in the past that executes without locking. A client can either specify a timestamp for a snapshot read, or provide an upper bound on the desired timestamp's staleness and let Spanner choose a timestamp. In either case, the execution of a snapshot read proceeds at any replica that is sufficiently up-to-date.

For both read-only transactions and snapshot reads, commit is inevitable once a timestamp has been chosen, unless the data at that timestamp has been garbage-collected. As a result, clients can avoid buffering results inside a retry loop. When a server fails, clients can internally continue the query on a different server by repeating the timestamp and the current read position.

4.1.1 Paxos Leader Leases

Spanner's Paxos implementation uses timed leases to make leadership long-lived (10 seconds by default). A potential leader sends requests for timed *lease votes*; upon receiving a quorum of lease votes the leader knows it has a lease. A replica extends its lease vote implicitly on a successful write, and the leader requests lease-vote extensions if they are near expiration. Define a leader's *lease interval* as starting when it discovers it has a quorum of lease votes, and as ending when it no longer has a quorum of lease votes (because some have expired). Spanner depends on the following disjointness invariant: for each Paxos group, each Paxos leader's lease interval is disjoint from every other leader's. Appendix A describes how this invariant is enforced.

The Spanner implementation permits a Paxos leader to abdicate by releasing its slaves from their lease votes. To preserve the disjointness invariant, Spanner constrains when abdication is permissible. Define s_{max} to be the maximum timestamp used by a leader. Subsequent sections will describe when s_{max} is advanced. Before abdicating, a leader must wait until $TT.after(s_{max})$ is true.

4.1.2 Assigning Timestamps to RW Transactions

Transactional reads and writes use two-phase locking. As a result, they can be assigned timestamps at any time

when all locks have been acquired, but before any locks have been released. For a given transaction, Spanner assigns it the timestamp that Paxos assigns to the Paxos write that represents the transaction commit.

Spanner depends on the following monotonicity invariant: within each Paxos group, Spanner assigns timestamps to Paxos writes in monotonically increasing order, even across leaders. A single leader replica can trivially assign timestamps in monotonically increasing order. This invariant is enforced across leaders by making use of the disjointness invariant: a leader must only assign timestamps within the interval of its leader lease. Note that whenever a timestamp s is assigned, s_{max} is advanced to s to preserve disjointness.

Spanner also enforces the following external-consistency invariant: if the start of a transaction T_2 occurs after the commit of a transaction T_1, then the commit timestamp of T_2 must be greater than the commit timestamp of T_1. Define the start and commit events for a transaction T_i by e_i^{start} and e_i^{commit}; and the commit timestamp of a transaction T_i by s_i. The invariant becomes $t_{abs}(e_1^{commit}) < t_{abs}(e_2^{start}) \Rightarrow s_1 < s_2$. The protocol for executing transactions and assigning timestamps obeys two rules, which together guarantee this invariant, as shown below. Define the arrival event of the commit request at the coordinator leader for a write T_i to be e_i^{server}.

Start The coordinator leader for a write T_i assigns a commit timestamp s_i no less than the value of *TT.now*().*latest*, computed after e_i^{server}. Note that the participant leaders do not matter here; Section 4.2.1 describes how they are involved in the implementation of the next rule.

Commit Wait The coordinator leader ensures that clients cannot see any data committed by T_i until *TT.after*(s_i) is true. Commit wait ensures that s_i is less than the absolute commit time of T_i, or $s_i < t_{abs}(e_i^{commit})$. The implementation of commit wait is described in Section 4.2.1. Proof:

$$
\begin{aligned}
s_1 &< t_{abs}(e_1^{commit}) && \text{(commit wait)}\\
t_{abs}(e_1^{commit}) &< t_{abs}(e_2^{start}) && \text{(assumption)}\\
t_{abs}(e_2^{start}) &\leq t_{abs}(e_2^{server}) && \text{(causality)}\\
t_{abs}(e_2^{server}) &\leq s_2 && \text{(start)}\\
s_1 &< s_2 && \text{(transitivity)}
\end{aligned}
$$

4.1.3 Serving Reads at a Timestamp

The monotonicity invariant described in Section 4.1.2 allows Spanner to correctly determine whether a replica's state is sufficiently up-to-date to satisfy a read. Every replica tracks a value called *safe time* t_{safe} which is the maximum timestamp at which a replica is up-to-date. A replica can satisfy a read at a timestamp t if $t <= t_{safe}$.

Define $t_{safe} = min(t_{safe}^{Paxos}, t_{safe}^{TM})$, where each Paxos state machine has a safe time t_{safe}^{Paxos} and each transaction manager has a safe time t_{safe}^{TM}. t_{safe}^{Paxos} is simpler: it is the timestamp of the highest-applied Paxos write. Because timestamps increase monotonically and writes are applied in order, writes will no longer occur at or below t_{safe}^{Paxos} with respect to Paxos.

t_{safe}^{TM} is ∞ at a replica if there are zero prepared (but not committed) transactions—that is, transactions in between the two phases of two-phase commit. (For a participant slave, t_{safe}^{TM} actually refers to the replica's leader's transaction manager, whose state the slave can infer through metadata passed on Paxos writes.) If there are any such transactions, then the state affected by those transactions is indeterminate: a participant replica does not know yet whether such transactions will commit. As we discuss in Section 4.2.1, the commit protocol ensures that every participant knows a lower bound on a prepared transaction's timestamp. Every participant leader (for a group g) for a transaction T_i assigns a prepare timestamp $s_{i,g}^{prepare}$ to its prepare record. The coordinator leader ensures that the transaction's commit timestamp $s_i >= s_{i,g}^{prepare}$ over all participant groups g. Therefore, for every replica in a group g, over all transactions T_i prepared at g, $t_{safe}^{TM} = min_i(s_{i,g}^{prepare}) - 1$ over all transactions prepared at g.

4.1.4 Assigning Timestamps to RO Transactions

A read-only transaction executes in two phases: assign a timestamp s_{read} [8], and then execute the transaction's reads as snapshot reads at s_{read}. The snapshot reads can execute at any replicas that are sufficiently up-to-date.

The simple assignment of $s_{read} =$ *TT.now*().*latest*, at any time after a transaction starts, preserves external consistency by an argument analogous to that presented for writes in Section 4.1.2. However, such a timestamp may require the execution of the data reads at s_{read} to block if t_{safe} has not advanced sufficiently. (In addition, note that choosing a value of s_{read} may also advance s_{max} to preserve disjointness.) To reduce the chances of blocking, Spanner should assign the oldest timestamp that preserves external consistency. Section 4.2.2 explains how such a timestamp can be chosen.

4.2 Details

This section explains some of the practical details of read-write transactions and read-only transactions elided earlier, as well as the implementation of a special transaction type used to implement atomic schema changes.

It then describes some refinements of the basic schemes as described.

4.2.1 Read-Write Transactions

Like Bigtable, writes that occur in a transaction are buffered at the client until commit. As a result, reads in a transaction do not see the effects of the transaction's writes. This design works well in Spanner because a read returns the timestamps of any data read, and uncommitted writes have not yet been assigned timestamps.

Reads within read-write transactions use wound-wait [33] to avoid deadlocks. The client issues reads to the leader replica of the appropriate group, which acquires read locks and then reads the most recent data. While a client transaction remains open, it sends keepalive messages to prevent participant leaders from timing out its transaction. When a client has completed all reads and buffered all writes, it begins two-phase commit. The client chooses a coordinator group and sends a commit message to each participant's leader with the identity of the coordinator and any buffered writes. Having the client drive two-phase commit avoids sending data twice across wide-area links.

A non-coordinator-participant leader first acquires write locks. It then chooses a prepare timestamp that must be larger than any timestamps it has assigned to previous transactions (to preserve monotonicity), and logs a prepare record through Paxos. Each participant then notifies the coordinator of its prepare timestamp.

The coordinator leader also first acquires write locks, but skips the prepare phase. It chooses a timestamp for the entire transaction after hearing from all other participant leaders. The commit timestamp s must be greater or equal to all prepare timestamps (to satisfy the constraints discussed in Section 4.1.3), greater than $TT.now().latest$ at the time the coordinator received its commit message, and greater than any timestamps the leader has assigned to previous transactions (again, to preserve monotonicity). The coordinator leader then logs a commit record through Paxos (or an abort if it timed out while waiting on the other participants).

Before allowing any coordinator replica to apply the commit record, the coordinator leader waits until $TT.after(s)$, so as to obey the commit-wait rule described in Section 4.1.2. Because the coordinator leader chose s based on $TT.now().latest$, and now waits until that timestamp is guaranteed to be in the past, the expected wait is at least $2 * \bar{\epsilon}$. This wait is typically overlapped with Paxos communication. After commit wait, the coordinator sends the commit timestamp to the client and all other participant leaders. Each participant leader logs the transaction's outcome through Paxos. All participants apply at the same timestamp and then release locks.

4.2.2 Read-Only Transactions

Assigning a timestamp requires a negotiation phase between all of the Paxos groups that are involved in the reads. As a result, Spanner requires a *scope* expression for every read-only transaction, which is an expression that summarizes the keys that will be read by the entire transaction. Spanner automatically infers the scope for standalone queries.

If the scope's values are served by a single Paxos group, then the client issues the read-only transaction to that group's leader. (The current Spanner implementation only chooses a timestamp for a read-only transaction at a Paxos leader.) That leader assigns s_{read} and executes the read. For a single-site read, Spanner generally does better than $TT.now().latest$. Define $LastTS()$ to be the timestamp of the last committed write at a Paxos group. If there are no prepared transactions, the assignment $s_{read} = LastTS()$ trivially satisfies external consistency: the transaction will see the result of the last write, and therefore be ordered after it.

If the scope's values are served by multiple Paxos groups, there are several options. The most complicated option is to do a round of communication with all of the groups's leaders to negotiate s_{read} based on $LastTS()$. Spanner currently implements a simpler choice. The client avoids a negotiation round, and just has its reads execute at $s_{read} = TT.now().latest$ (which may wait for safe time to advance). All reads in the transaction can be sent to replicas that are sufficiently up-to-date.

4.2.3 Schema-Change Transactions

TrueTime enables Spanner to support atomic schema changes. It would be infeasible to use a standard transaction, because the number of participants (the number of groups in a database) could be in the millions. Bigtable supports atomic schema changes in one datacenter, but its schema changes block all operations.

A Spanner schema-change transaction is a generally non-blocking variant of a standard transaction. First, it is explicitly assigned a timestamp in the future, which is registered in the prepare phase. As a result, schema changes across thousands of servers can complete with minimal disruption to other concurrent activity. Second, reads and writes, which implicitly depend on the schema, synchronize with any registered schema-change timestamp at time t: they may proceed if their timestamps precede t, but they must block behind the schema-change transaction if their timestamps are after t. Without TrueTime, defining the schema change to happen at t would be meaningless.

replicas	latency (ms)			throughput (Kops/sec)		
	write	read-only transaction	snapshot read	write	read-only transaction	snapshot read
1D	9.4±.6	—	—	4.0±.3	—	—
1	14.4±1.0	1.4±.1	1.3±.1	4.1±.05	10.9±.4	13.5±.1
3	13.9±.6	1.3±.1	1.2±.1	2.2±.5	13.8±3.2	38.5±.3
5	14.4±.4	1.4±.05	1.3±.04	2.8±.3	25.3±5.2	50.0±1.1

Table 3: Operation microbenchmarks. Mean and standard deviation over 10 runs. 1D means one replica with commit wait disabled.

4.2.4 Refinements

t_{safe}^{TM} as defined above has a weakness, in that a single prepared transaction prevents t_{safe} from advancing. As a result, no reads can occur at later timestamps, even if the reads do not conflict with the transaction. Such false conflicts can be removed by augmenting t_{safe}^{TM} with a fine-grained mapping from key ranges to prepared-transaction timestamps. This information can be stored in the lock table, which already maps key ranges to lock metadata. When a read arrives, it only needs to be checked against the fine-grained safe time for key ranges with which the read conflicts.

LastTS() as defined above has a similar weakness: if a transaction has just committed, a non-conflicting read-only transaction must still be assigned s_{read} so as to follow that transaction. As a result, the execution of the read could be delayed. This weakness can be remedied similarly by augmenting *LastTS*() with a fine-grained mapping from key ranges to commit timestamps in the lock table. (We have not yet implemented this optimization.) When a read-only transaction arrives, its timestamp can be assigned by taking the maximum value of *LastTS*() for the key ranges with which the transaction conflicts, unless there is a conflicting prepared transaction (which can be determined from fine-grained safe time).

t_{safe}^{Paxos} as defined above has a weakness in that it cannot advance in the absence of Paxos writes. That is, a snapshot read at t cannot execute at Paxos groups whose last write happened before t. Spanner addresses this problem by taking advantage of the disjointness of leader-lease intervals. Each Paxos leader advances t_{safe}^{Paxos} by keeping a threshold above which future writes' timestamps will occur: it maintains a mapping *MinNextTS*(n) from Paxos sequence number n to the minimum timestamp that may be assigned to Paxos sequence number $n + 1$. A replica can advance t_{safe}^{Paxos} to *MinNextTS*$(n) - 1$ when it has applied through n.

A single leader can enforce its *MinNextTS*() promises easily. Because the timestamps promised by *MinNextTS*() lie within a leader's lease, the disjointness invariant enforces *MinNextTS*() promises across leaders. If a leader wishes to advance *MinNextTS*() beyond the end of its leader lease, it must first extend its lease. Note that s_{max} is always advanced to the highest value in *MinNextTS*() to preserve disjointness.

A leader by default advances *MinNextTS*() values every 8 seconds. Thus, in the absence of prepared transactions, healthy slaves in an idle Paxos group can serve reads at timestamps greater than 8 seconds old in the worst case. A leader may also advance *MinNextTS*() values on demand from slaves.

5 Evaluation

We first measure Spanner's performance with respect to replication, transactions, and availability. We then provide some data on TrueTime behavior, and a case study of our first client, F1.

5.1 Microbenchmarks

Table 3 presents some microbenchmarks for Spanner. These measurements were taken on timeshared machines: each spanserver ran on scheduling units of 4GB RAM and 4 cores (AMD Barcelona 2200MHz). Clients were run on separate machines. Each zone contained one spanserver. Clients and zones were placed in a set of datacenters with network distance of less than 1ms. (Such a layout should be commonplace: most applications do not need to distribute all of their data worldwide.) The test database was created with 50 Paxos groups with 2500 directories. Operations were standalone reads and writes of 4KB. All reads were served out of memory after a compaction, so that we are only measuring the overhead of Spanner's call stack. In addition, one unmeasured round of reads was done first to warm any location caches.

For the latency experiments, clients issued sufficiently few operations so as to avoid queuing at the servers. From the 1-replica experiments, commit wait is about 5ms, and Paxos latency is about 9ms. As the number of replicas increases, the latency stays roughly constant with less standard deviation because Paxos executes in parallel at a group's replicas. As the number of replicas increases, the latency to achieve a quorum becomes less sensitive to slowness at one slave replica.

For the throughput experiments, clients issued sufficiently many operations so as to saturate the servers'

participants	latency (ms)	
	mean	99th percentile
1	17.0 ±1.4	75.0 ±34.9
2	24.5 ±2.5	87.6 ±35.9
5	31.5 ±6.2	104.5 ±52.2
10	30.0 ±3.7	95.6 ±25.4
25	35.5 ±5.6	100.4 ±42.7
50	42.7 ±4.1	93.7 ±22.9
100	71.4 ±7.6	131.2 ±17.6
200	150.5 ±11.0	320.3 ±35.1

Table 4: Two-phase commit scalability. Mean and standard deviations over 10 runs.

CPUs. Snapshot reads can execute at any up-to-date replicas, so their throughput increases almost linearly with the number of replicas. Single-read read-only transactions only execute at leaders because timestamp assignment must happen at leaders. Read-only-transaction throughput increases with the number of replicas because the number of effective spanservers increases: in the experimental setup, the number of spanservers equaled the number of replicas, and leaders were randomly distributed among the zones. Write throughput benefits from the same experimental artifact (which explains the increase in throughput from 3 to 5 replicas), but that benefit is outweighed by the linear increase in the amount of work performed per write, as the number of replicas increases.

Table 4 demonstrates that two-phase commit can scale to a reasonable number of participants: it summarizes a set of experiments run across 3 zones, each with 25 spanservers. Scaling up to 50 participants is reasonable in both mean and 99th-percentile, and latencies start to rise noticeably at 100 participants.

5.2 Availability

Figure 5 illustrates the availability benefits of running Spanner in multiple datacenters. It shows the results of three experiments on throughput in the presence of datacenter failure, all of which are overlaid onto the same time scale. The test universe consisted of 5 zones Z_i, each of which had 25 spanservers. The test database was sharded into 1250 Paxos groups, and 100 test clients constantly issued non-snapshot reads at an aggregate rate of 50K reads/second. All of the leaders were explicitly placed in Z_1. Five seconds into each test, all of the servers in one zone were killed: *non-leader* kills Z_2; *leader-hard* kills Z_1; *leader-soft* kills Z_1, but it gives notifications to all of the servers that they should handoff leadership first.

Killing Z_2 has no effect on read throughput. Killing Z_1 while giving the leaders time to handoff leadership to

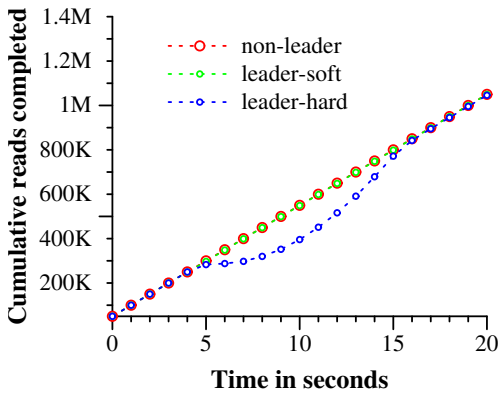


Figure 5: Effect of killing servers on throughput.

a different zone has a minor effect: the throughput drop is not visible in the graph, but is around 3-4%. On the other hand, killing Z_1 with no warning has a severe effect: the rate of completion drops almost to 0. As leaders get re-elected, though, the throughput of the system rises to approximately 100K reads/second because of two artifacts of our experiment: there is extra capacity in the system, and operations are queued while the leader is unavailable. As a result, the throughput of the system rises before leveling off again at its steady-state rate.

We can also see the effect of the fact that Paxos leader leases are set to 10 seconds. When we kill the zone, the leader-lease expiration times for the groups should be evenly distributed over the next 10 seconds. Soon after each lease from a dead leader expires, a new leader is elected. Approximately 10 seconds after the kill time, all of the groups have leaders and throughput has recovered. Shorter lease times would reduce the effect of server deaths on availability, but would require greater amounts of lease-renewal network traffic. We are in the process of designing and implementing a mechanism that will cause slaves to release Paxos leader leases upon leader failure.

5.3 TrueTime

Two questions must be answered with respect to TrueTime: is ϵ truly a bound on clock uncertainty, and how bad does ϵ get? For the former, the most serious problem would be if a local clock's drift were greater than 200us/sec: that would break assumptions made by TrueTime. Our machine statistics show that bad CPUs are 6 times more likely than bad clocks. That is, clock issues are extremely infrequent, relative to much more serious hardware problems. As a result, we believe that TrueTime's implementation is as trustworthy as any other piece of software upon which Spanner depends.

Figure 6 presents TrueTime data taken at several thousand spanserver machines across datacenters up to 2200

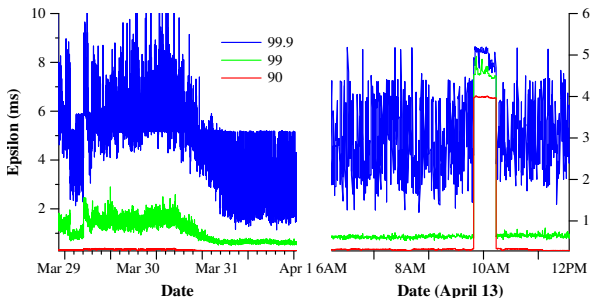


Figure 6: Distribution of TrueTime ϵ values, sampled right after timeslave daemon polls the time masters. 90th, 99th, and 99.9th percentiles are graphed.

km apart. It plots the 90th, 99th, and 99.9th percentiles of ϵ, sampled at timeslave daemons immediately after polling the time masters. This sampling elides the sawtooth in ϵ due to local-clock uncertainty, and therefore measures time-master uncertainty (which is generally 0) plus communication delay to the time masters.

The data shows that these two factors in determining the base value of ϵ are generally not a problem. However, there can be significant tail-latency issues that cause higher values of ϵ. The reduction in tail latencies beginning on March 30 were due to networking improvements that reduced transient network-link congestion. The increase in ϵ on April 13, approximately one hour in duration, resulted from the shutdown of 2 time masters at a datacenter for routine maintenance. We continue to investigate and remove causes of TrueTime spikes.

5.4 F1

Spanner started being experimentally evaluated under production workloads in early 2011, as part of a rewrite of Google's advertising backend called F1 [35]. This backend was originally based on a MySQL database that was manually sharded many ways. The uncompressed dataset is tens of terabytes, which is small compared to many NoSQL instances, but was large enough to cause difficulties with sharded MySQL. The MySQL sharding scheme assigned each customer and all related data to a fixed shard. This layout enabled the use of indexes and complex query processing on a per-customer basis, but required some knowledge of the sharding in application business logic. Resharding this revenue-critical database as it grew in the number of customers and their data was extremely costly. The last resharding took over two years of intense effort, and involved coordination and testing across dozens of teams to minimize risk. This operation was too complex to do regularly: as a result, the team had to limit growth on the MySQL database by storing some data in external Bigtables, which compromised transactional behavior and the ability to query across all data.

# fragments	# directories
1	>100M
2–4	341
5–9	5336
10–14	232
15–99	34
100–500	7

Table 5: Distribution of directory-fragment counts in F1.

The F1 team chose to use Spanner for several reasons. First, Spanner removes the need to manually reshard. Second, Spanner provides synchronous replication and automatic failover. With MySQL master-slave replication, failover was difficult, and risked data loss and downtime. Third, F1 requires strong transactional semantics, which made using other NoSQL systems impractical. Application semantics requires transactions across arbitrary data, and consistent reads. The F1 team also needed secondary indexes on their data (since Spanner does not yet provide automatic support for secondary indexes), and was able to implement their own consistent global indexes using Spanner transactions.

All application writes are now by default sent through F1 to Spanner, instead of the MySQL-based application stack. F1 has 2 replicas on the west coast of the US, and 3 on the east coast. This choice of replica sites was made to cope with outages due to potential major natural disasters, and also the choice of their frontend sites. Anecdotally, Spanner's automatic failover has been nearly invisible to them. Although there have been unplanned cluster failures in the last few months, the most that the F1 team has had to do is update their database's schema to tell Spanner where to preferentially place Paxos leaders, so as to keep them close to where their frontends moved.

Spanner's timestamp semantics made it efficient for F1 to maintain in-memory data structures computed from the database state. F1 maintains a logical history log of all changes, which is written into Spanner itself as part of every transaction. F1 takes full snapshots of data at a timestamp to initialize its data structures, and then reads incremental changes to update them.

Table 5 illustrates the distribution of the number of fragments per directory in F1. Each directory typically corresponds to a customer in the application stack above F1. The vast majority of directories (and therefore customers) consist of only 1 fragment, which means that reads and writes to those customers' data are guaranteed to occur on only a single server. The directories with more than 100 fragments are all tables that contain F1 secondary indexes: writes to more than a few fragments

operation	latency (ms)		count
	mean	std dev	
all reads	8.7	376.4	21.5B
single-site commit	72.3	112.8	31.2M
multi-site commit	103.0	52.2	32.1M

Table 6: F1-perceived operation latencies measured over the course of 24 hours.

of such tables are extremely uncommon. The F1 team has only seen such behavior when they do untuned bulk data loads as transactions.

Table 6 presents Spanner operation latencies as measured from F1 servers. Replicas in the east-coast data centers are given higher priority in choosing Paxos leaders. The data in the table is measured from F1 servers in those data centers. The large standard deviation in write latencies is caused by a pretty fat tail due to lock conflicts. The even larger standard deviation in read latencies is partially due to the fact that Paxos leaders are spread across two data centers, only one of which has machines with SSDs. In addition, the measurement includes every read in the system from two datacenters: the mean and standard deviation of the bytes read were roughly 1.6KB and 119KB, respectively.

6 Related Work

Consistent replication across datacenters as a storage service has been provided by Megastore [5] and DynamoDB [3]. DynamoDB presents a key-value interface, and only replicates within a region. Spanner follows Megastore in providing a semi-relational data model, and even a similar schema language. Megastore does not achieve high performance. It is layered on top of Bigtable, which imposes high communication costs. It also does not support long-lived leaders: multiple replicas may initiate writes. All writes from different replicas necessarily conflict in the Paxos protocol, even if they do not logically conflict: throughput collapses on a Paxos group at several writes per second. Spanner provides higher performance, general-purpose transactions, and external consistency.

Pavlo et al. [31] have compared the performance of databases and MapReduce [12]. They point to several other efforts that have been made to explore database functionality layered on distributed key-value stores [1, 4, 7, 41] as evidence that the two worlds are converging. We agree with the conclusion, but demonstrate that integrating multiple layers has its advantages: integrating concurrency control with replication reduces the cost of commit wait in Spanner, for example.

The notion of layering transactions on top of a replicated store dates at least as far back as Gifford's dissertation [16]. Scatter [17] is a recent DHT-based key-value store that layers transactions on top of consistent replication. Spanner focuses on providing a higher-level interface than Scatter does. Gray and Lamport [18] describe a non-blocking commit protocol based on Paxos. Their protocol incurs more messaging costs than two-phase commit, which would aggravate the cost of commit over widely distributed groups. Walter [36] provides a variant of snapshot isolation that works within, but not across datacenters. In contrast, our read-only transactions provide a more natural semantics, because we support external consistency over all operations.

There has been a spate of recent work on reducing or eliminating locking overheads. Calvin [40] eliminates concurrency control: it pre-assigns timestamps and then executes the transactions in timestamp order. H-Store [39] and Granola [11] each supported their own classification of transaction types, some of which could avoid locking. None of these systems provides external consistency. Spanner addresses the contention issue by providing support for snapshot isolation.

VoltDB [42] is a sharded in-memory database that supports master-slave replication over the wide area for disaster recovery, but not more general replication configurations. It is an example of what has been called NewSQL, which is a marketplace push to support scalable SQL [38]. A number of commercial databases implement reads in the past, such as MarkLogic [26] and Oracle's Total Recall [30]. Lomet and Li [24] describe an implementation strategy for such a temporal database.

Farsite derived bounds on clock uncertainty (much looser than TrueTime's) relative to a trusted clock reference [13]: server leases in Farsite were maintained in the same way that Spanner maintains Paxos leases. Loosely synchronized clocks have been used for concurrency-control purposes in prior work [2, 23]. We have shown that TrueTime lets one reason about global time across sets of Paxos state machines.

7 Future Work

We have spent most of the last year working with the F1 team to transition Google's advertising backend from MySQL to Spanner. We are actively improving its monitoring and support tools, as well as tuning its performance. In addition, we have been working on improving the functionality and performance of our backup/restore system. We are currently implementing the Spanner schema language, automatic maintenance of secondary indices, and automatic load-based resharding. Longer term, there are a couple of features that we plan to in-

vestigate. Optimistically doing reads in parallel may be a valuable strategy to pursue, but initial experiments have indicated that the right implementation is non-trivial. In addition, we plan to eventually support direct changes of Paxos configurations [22, 34].

Given that we expect many applications to replicate their data across datacenters that are relatively close to each other, TrueTime ϵ may noticeably affect performance. We see no insurmountable obstacle to reducing ϵ below 1ms. Time-master-query intervals can be reduced, and better clock crystals are relatively cheap. Time-master query latency could be reduced with improved networking technology, or possibly even avoided through alternate time-distribution technology.

Finally, there are obvious areas for improvement. Although Spanner is scalable in the number of nodes, the node-local data structures have relatively poor performance on complex SQL queries, because they were designed for simple key-value accesses. Algorithms and data structures from DB literature could improve single-node performance a great deal. Second, moving data automatically between datacenters in response to changes in client load has long been a goal of ours, but to make that goal effective, we would also need the ability to move client-application processes between datacenters in an automated, coordinated fashion. Moving processes raises the even more difficult problem of managing resource acquisition and allocation between datacenters.

8 Conclusions

To summarize, Spanner combines and extends on ideas from two research communities: from the database community, a familiar, easy-to-use, semi-relational interface, transactions, and an SQL-based query language; from the systems community, scalability, automatic sharding, fault tolerance, consistent replication, external consistency, and wide-area distribution. Since Spanner's inception, we have taken more than 5 years to iterate to the current design and implementation. Part of this long iteration phase was due to a slow realization that Spanner should do more than tackle the problem of a globally-replicated namespace, and should also focus on database features that Bigtable was missing.

One aspect of our design stands out: the linchpin of Spanner's feature set is TrueTime. We have shown that reifying clock uncertainty in the time API makes it possible to build distributed systems with much stronger time semantics. In addition, as the underlying system enforces tighter bounds on clock uncertainty, the overhead of the stronger semantics decreases. As a community, we should no longer depend on loosely synchronized clocks and weak time APIs in designing distributed algorithms.

Acknowledgements

Many people have helped to improve this paper: our shepherd Jon Howell, who went above and beyond his responsibilities; the anonymous referees; and many Googlers: Atul Adya, Fay Chang, Frank Dabek, Sean Dorward, Bob Gruber, David Held, Nick Kline, Alex Thomson, and Joel Wein. Our management has been very supportive of both our work and of publishing this paper: Aristotle Balogh, Bill Coughran, Urs Hölzle, Doron Meyer, Cos Nicolaou, Kathy Polizzi, Sridhar Ramaswany, and Shivakumar Venkataraman.

We have built upon the work of the Bigtable and Megastore teams. The F1 team, and Jeff Shute in particular, worked closely with us in developing our data model and helped immensely in tracking down performance and correctness bugs. The Platforms team, and Luiz Barroso and Bob Felderman in particular, helped to make TrueTime happen. Finally, a lot of Googlers used to be on our team: Ken Ashcraft, Paul Cychosz, Krzysztof Ostrowski, Amir Voskoboynik, Matthew Weaver, Theo Vassilakis, and Eric Veach; or have joined our team recently: Nathan Bales, Adam Beberg, Vadim Borisov, Ken Chen, Brian Cooper, Cian Cullinan, Robert-Jan Huijsman, Milind Joshi, Andrey Khorlin, Dawid Kuroczko, Laramie Leavitt, Eric Li, Mike Mammarella, Sunil Mushran, Simon Nielsen, Ovidiu Platon, Ananth Shrinivas, Vadim Suvorov, and Marcel van der Holst.

References

[1] Azza Abouzeid et al. "HadoopDB: an architectural hybrid of MapReduce and DBMS technologies for analytical workloads". *Proc. of VLDB*. 2009, pp. 922–933.

[2] A. Adya et al. "Efficient optimistic concurrency control using loosely synchronized clocks". *Proc. of SIGMOD*. 1995, pp. 23–34.

[3] Amazon. *Amazon DynamoDB*. 2012.

[4] Michael Armbrust et al. "PIQL: Success-Tolerant Query Processing in the Cloud". *Proc. of VLDB*. 2011, pp. 181–192.

[5] Jason Baker et al. "Megastore: Providing Scalable, Highly Available Storage for Interactive Services". *Proc. of CIDR*. 2011, pp. 223–234.

[6] Hal Berenson et al. "A critique of ANSI SQL isolation levels". *Proc. of SIGMOD*. 1995, pp. 1–10.

[7] Matthias Brantner et al. "Building a database on S3". *Proc. of SIGMOD*. 2008, pp. 251–264.

[8] A. Chan and R. Gray. "Implementing Distributed Read-Only Transactions". *IEEE TOSE* SE-11.2 (Feb. 1985), pp. 205–212.

[9] Fay Chang et al. "Bigtable: A Distributed Storage System for Structured Data". *ACM TOCS* 26.2 (June 2008), 4:1–4:26.

[10] Brian F. Cooper et al. "PNUTS: Yahoo!'s hosted data serving platform". *Proc. of VLDB*. 2008, pp. 1277–1288.

[11] James Cowling and Barbara Liskov. "Granola: Low-Overhead Distributed Transaction Coordination". *Proc. of USENIX ATC*. 2012, pp. 223–236.

[12] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: a flexible data processing tool". *CACM* 53.1 (Jan. 2010), pp. 72–77.

[13] John Douceur and Jon Howell. *Scalable Byzantine-Fault-Quantifying Clock Synchronization*. Tech. rep. MSR-TR-2003-67. MS Research, 2003.

[14] John R. Douceur and Jon Howell. "Distributed directory service in the Farsite file system". *Proc. of OSDI*. 2006, pp. 321–334.

[15] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. "The Google file system". *Proc. of SOSP*. Dec. 2003, pp. 29–43.

[16] David K. Gifford. *Information Storage in a Decentralized Computer System*. Tech. rep. CSL-81-8. PhD dissertation. Xerox PARC, July 1982.

[17] Lisa Glendenning et al. "Scalable consistency in Scatter". *Proc. of SOSP*. 2011.

[18] Jim Gray and Leslie Lamport. "Consensus on transaction commit". *ACM TODS* 31.1 (Mar. 2006), pp. 133–160.

[19] Pat Helland. "Life beyond Distributed Transactions: an Apostate's Opinion". *Proc. of CIDR*. 2007, pp. 132–141.

[20] Maurice P. Herlihy and Jeannette M. Wing. "Linearizability: a correctness condition for concurrent objects". *ACM TOPLAS* 12.3 (July 1990), pp. 463–492.

[21] Leslie Lamport. "The part-time parliament". *ACM TOCS* 16.2 (May 1998), pp. 133–169.

[22] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. "Reconfiguring a state machine". *SIGACT News* 41.1 (Mar. 2010), pp. 63–73.

[23] Barbara Liskov. "Practical uses of synchronized clocks in distributed systems". *Distrib. Comput.* 6.4 (July 1993), pp. 211–219.

[24] David B. Lomet and Feifei Li. "Improving Transaction-Time DBMS Performance and Functionality". *Proc. of ICDE* (2009), pp. 581–591.

[25] Jacob R. Lorch et al. "The SMART way to migrate replicated stateful services". *Proc. of EuroSys*. 2006, pp. 103–115.

[26] MarkLogic. *MarkLogic 5 Product Documentation*. 2012.

[27] Keith Marzullo and Susan Owicki. "Maintaining the time in a distributed system". *Proc. of PODC*. 1983, pp. 295–305.

[28] Sergey Melnik et al. "Dremel: Interactive Analysis of Web-Scale Datasets". *Proc. of VLDB*. 2010, pp. 330–339.

[29] D.L. Mills. *Time synchronization in DCNET hosts*. Internet Project Report IEN–173. COMSAT Laboratories, Feb. 1981.

[30] Oracle. *Oracle Total Recall*. 2012.

[31] Andrew Pavlo et al. "A comparison of approaches to large-scale data analysis". *Proc. of SIGMOD*. 2009, pp. 165–178.

[32] Daniel Peng and Frank Dabek. "Large-scale incremental processing using distributed transactions and notifications". *Proc. of OSDI*. 2010, pp. 1–15.

[33] Daniel J. Rosenkrantz, Richard E. Stearns, and Philip M. Lewis II. "System level concurrency control for distributed database systems". *ACM TODS* 3.2 (June 1978), pp. 178–198.

[34] Alexander Shraer et al. "Dynamic Reconfiguration of Primary/Backup Clusters". *Proc. of USENIX ATC*. 2012, pp. 425–438.

[35] Jeff Shute et al. "F1 — The Fault-Tolerant Distributed RDBMS Supporting Google's Ad Business". *Proc. of SIGMOD*. May 2012, pp. 777–778.

[36] Yair Sovran et al. "Transactional storage for geo-replicated systems". *Proc. of SOSP*. 2011, pp. 385–400.

[37] Michael Stonebraker. *Why Enterprises Are Uninterested in NoSQL*. 2010.

[38] Michael Stonebraker. *Six SQL Urban Myths*. 2010.

[39] Michael Stonebraker et al. "The end of an architectural era: (it's time for a complete rewrite)". *Proc. of VLDB*. 2007, pp. 1150–1160.

[40] Alexander Thomson et al. "Calvin: Fast Distributed Transactions for Partitioned Database Systems". *Proc. of SIGMOD*. 2012, pp. 1–12.

[41] Ashish Thusoo et al. "Hive — A Petabyte Scale Data Warehouse Using Hadoop". *Proc. of ICDE*. 2010, pp. 996–1005.

[42] VoltDB. *VoltDB Resources*. 2012.

A Paxos Leader-Lease Management

The simplest means to ensure the disjointness of Paxos-leader-lease intervals would be for a leader to issue a synchronous Paxos write of the lease interval, whenever it would be extended. A subsequent leader would read the interval and wait until that interval has passed.

TrueTime can be used to ensure disjointness without these extra log writes. The potential ith leader keeps a lower bound on the start of a lease vote from replica r as $v_{i,r}^{leader} = TT.now().earliest$, computed before $e_{i,r}^{send}$ (defined as when the lease request is sent by the leader). Each replica r grants a lease at lease $e_{i,r}^{grant}$, which happens after $e_{i,r}^{receive}$ (when the replica receives a lease request); the lease ends at $t_{i,r}^{end} = TT.now().latest + 10$, computed after $e_{i,r}^{receive}$. A replica r obeys the **single-vote** rule: it will not grant another lease vote until $TT.after(t_{i,r}^{end})$ is true. To enforce this rule across different incarnations of r, Spanner logs a lease vote at the granting replica before granting the lease; this log write can be piggybacked upon existing Paxos-protocol log writes.

When the ith leader receives a quorum of votes (event e_i^{quorum}), it computes its lease interval as $lease_i = [TT.now().latest, min_r(v_{i,r}^{leader}) + 10]$. The lease is deemed to have expired at the leader when $TT.before(min_r(v_{i,r}^{leader}) + 10)$ is false. To prove disjointness, we make use of the fact that the ith and $(i+1)$th leaders must have one replica in common in their quorums. Call that replica $r0$. Proof:

$$
\begin{aligned}
lease_i.end &= min_r(v_{i,r}^{leader}) + 10 && \text{(by definition)}\\
min_r(v_{i,r}^{leader}) + 10 &\le v_{i,r0}^{leader} + 10 && \text{(min)}\\
v_{i,r0}^{leader} + 10 &\le t_{abs}(e_{i,r0}^{send}) + 10 && \text{(by definition)}\\
t_{abs}(e_{i,r0}^{send}) + 10 &\le t_{abs}(e_{i,r0}^{receive}) + 10 && \text{(causality)}\\
t_{abs}(e_{i,r0}^{receive}) + 10 &\le t_{i,r0}^{end} && \text{(by definition)}\\
t_{i,r0}^{end} &< t_{abs}(e_{i+1,r0}^{grant}) && \text{(single-vote)}\\
t_{abs}(e_{i+1,r0}^{grant}) &\le t_{abs}(e_{i+1}^{quorum}) && \text{(causality)}\\
t_{abs}(e_{i+1}^{quorum}) &\le lease_{i+1}.start && \text{(by definition)}
\end{aligned}
$$

Chapter 17

Spanner, TrueTime and the CAP Theorem

Authors:
Eric Brewer

Publication year: 2017

Spanner, TrueTime & The CAP Theorem

Eric Brewer
VP, Infrastructure, Google

February 14, 2017

Spanner is Google's highly available global SQL database [CDE+12]. It manages replicated data at great scale, both in terms of size of data and volume of transactions. It assigns globally consistent real-time timestamps to every datum written to it, and clients can do globally consistent reads across the entire database without locking.

The CAP theorem [Bre12] says that you can only have two of the three desirable properties of:

- C: Consistency, which we can think of as serializability for this discussion;
- A: 100% availability, for both reads and updates;
- P: tolerance to network partitions.

This leads to three kinds of systems: CA, CP and AP, based on what letter you leave out. Note that you are not entitled to 2 of 3, and many systems have zero or one of the properties.

For distributed systems over a "wide area", it is generally viewed that partitions are inevitable, although not necessarily common [BK14]. Once you believe that partitions are inevitable, any distributed system must be prepared to forfeit either consistency (AP) or availability (CP), which is not a choice anyone wants to make. In fact, the original point of the CAP theorem was to get designers to take this tradeoff seriously. But there are two important caveats: first, you only need forfeit something during an actual partition, and even then there are many mitigations (see the "12 years" paper [Bre12]). Second, the actual theorem is about 100% availability, while the interesting discussion here is about the tradeoffs involved for realistic high availability.

Spanner claims to be consistent and available

Despite being a global distributed system, Spanner claims to be consistent and highly available, which implies there are no partitions and thus many are skeptical.[1] Does this mean that Spanner is a CA system as defined by CAP? The short answer is "no" technically, but "yes" in effect and its users can and do assume CA.

The purist answer is "no" because partitions can happen and in fact have happened at Google, and during (some) partitions, Spanner chooses C and forfeits A. It is technically a CP system. We explore the impact of partitions below.

Given that Spanner always provides consistency, the real question for a claim of CA is whether or not Spanner's serious users assume its availability. If its actual availability is so high that users can ignore outages, then Spanner can justify an "effectively CA" claim. This does not imply 100% availability (and Spanner does not and will not provide it), but rather something like 5 or more "9s" (1 failure in 10^5 or less). In turn, the real litmus test is whether or not users (that want their own service to be highly available)

[1] Although I work for Google, I have not worked on Spanner or TrueTime, other than to push to make both available to our Cloud customers. My intent is to provide an objective outsider's view of Spanner, but I admit bias in that I like the system in addition to working for Google.

write the code to handle outage exceptions: if they haven't written that code, then they are assuming high availability. Based on a large number of internal users of Spanner, we know that they assume Spanner is highly available.

A second refinement is that there are many other sources of outages, some of which take out the users in addition to Spanner ("fate sharing"). We actually care about the differential availability, in which the user is up (and making a request) to notice that Spanner is down. This number is strictly higher (more available) than Spanner's actual availability — that is, you have to hear the tree fall to count it as a problem.

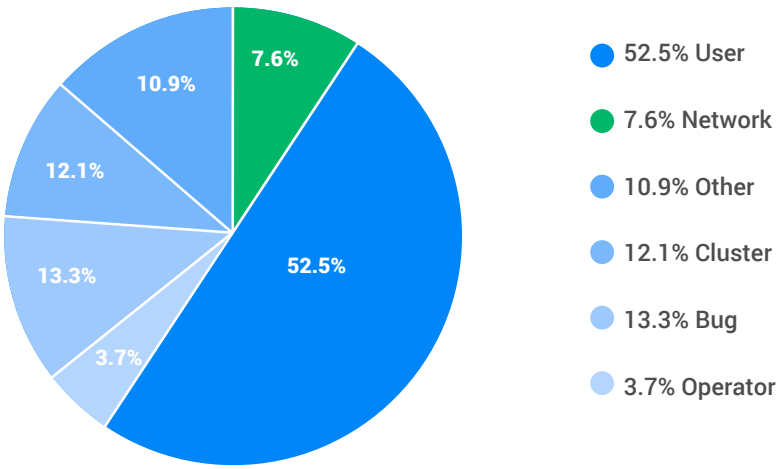
A third issue is whether or not outages are due to partitions. If the primary causes of Spanner outages are not partitions, then CA is in some sense more accurate. For example, any database cannot provide availability if all of its replicas are offline, which has nothing to do with partitions. Such a multi-replica outage should be very rare, but if partitions are significantly more rare, then you can effectively ignore partitions as a factor in availability. For Spanner, this means that when there is an availability outage, it is not in practice due to a partition, but rather some other set of multiple faults (as no single fault will forfeit availability).

Availability data

Before we get to Spanner, it is worth taking a look at the evolution of Chubby, another wide-area system that provides both consistency and availability. The original Chubby paper [Bur06] mentioned nine outages of 30 seconds or more in 700 days, and six of those were network related (as discussed in [BK14]). This corresponds to an availability worse than 5 9s (at best), to a more realistic 4 9s if we assume an average of 10 minutes per outage, and potentially even 3 9s at hours per outage.

For locking and consistent read/write operations, **modern geographically distributed Chubby cells provide an average availability of 99.99958%** (for 30s+ outages) due to various network, architectural and operational improvements. Starting in 2009, due to "excess" availability, Chubby's Site Reliability Engineers (SREs) started forcing periodic outages to ensure we continue to understand dependencies and the impact of Chubby failures.

Internally, Spanner provides a similar level of reliability to Chubby; that is, better than 5 9s. The Cloud version has the same foundation, but adds some new pieces, so it may be a little lower in practice for a while.



The pie chart above reveals the causes of Spanner incidents internally. An incident is an unexpected event, but not all incidents are outages; some can be masked easily. The chart is weighted by frequency not by impact. The bulk of the incidents **(User)** are due to user errors such as overload or misconfiguration and mostly affect that user, whereas the remaining categories could affect all users in an area. **Cluster** incidents reflect non-network problems in the underlying infrastructure, including problems with servers and power. Spanner automatically works around these incidents by using other replicas; however, SRE involvement is sometimes required to fix a broken replica. **Operator** incidents are accidents induced by SREs, such as a misconfiguration. **Bug** implies a software error that caused some problem; these can lead to large or small outages. The two biggest outages were both due to software bugs that affected all replicas of a particular database at the same time. **Other** is grab bag of various problems, most of which occurred only once.

The **Network** category, under 8%, is where partitions and networking configuration problems appear. There were no events in which a large set of clusters were partitioned from another large set of clusters. Nor was a Spanner quorum ever on the minority side of a partition. We did see individual data centers or regions get cut off from the rest of the network. We also had some misconfigurations that under-provisioned bandwidth temporarily, and we saw some temporary periods of bad latency related to hardware failures. We saw one issue in which one direction of traffic failed, causing a weird partition that had to be resolved by bringing down some nodes. So far, no large outages were due to networking incidents.

Summarizing, to claim "effectively CA" a system must be in this state of relative probabilities: 1) At a minimum it must have very high availability in practice (so that users can ignore exceptions), and 2) as this is about partitions it should also have a low fraction of those outages due to partitions. Spanner meets both.

It's the network

Many assume that Spanner somehow gets around CAP via its use of TrueTime, which is a service that enables the use of globally synchronized clocks. Although remarkable, TrueTime does not significantly help achieve CA; its actual value is covered below. To the extent there is anything special, it is really Google's wide-area network, plus many years of operational improvements, that greatly limit partitions in practice, and thus enable high availability.

First, Google runs its own private global network. Spanner is not running over the public Internet — in fact, every Spanner packet flows only over Google-controlled routers and links (excluding any edge links to remote clients). Furthermore, each data center typically has at least three independent fibers connecting it to the private global network, thus ensuring path diversity for every pair of data centers.[2] Similarly, there is redundancy of equipment and paths within a datacenter. Thus normally catastrophic events, such as cut fiber lines, do not lead to partitions or to outages.

The real risk for a partition is thus not a cut path, but rather some kind of broad config or software upgrade that breaks multiple paths simultaneously. This is a real risk and something that Google continues to work to prevent and mitigate. The general strategy is to limit the impact (or "blast radius") of any particular update, so that when we inevitably push a bad change, it only takes out some paths or some replicas. We then fix those before attempting any other changes.

[2]The actual requirement is a target availability, not a number of connections per se.

Although the network can greatly reduce partitions, it cannot improve the speed of light. Consistent operations that span a wide area have a significant minimum round trip time, which can be tens of milliseconds or more across continents. (A distance of 1000 miles is about 5 million feet, so at ½ foot per nanosecond, the minimum would be 10 ms.) Google defines "regions" to have a 2ms round-trip time, so that regional offerings provide a balance between latency and disaster tolerance. Spanner mitigates latency via extensive pipelining of transactions, but that does not help single-transaction latency. For reads, latency is typically low, due to global timestamps and the ability to use a local replica (covered below).

A model with weaker consistency could have lower update latency. However, without the long round trip it would also have a window of lower durability, since a disaster could take out the local site and delete all copies before the data is replicated to another region.

What happens during a Partition

To understand partitions, we need to know a little bit more about how Spanner works. As with most ACID databases, Spanner uses two-phase commit (2PC) and strict two-phase locking to ensure isolation and strong consistency. 2PC has been called the "anti-availability" protocol [Hel16] because all members must be up for it to work. Spanner mitigates this by having each member be a Paxos group, thus ensuring each 2PC "member" is highly available even if some of its Paxos participants are down. Data is divided into groups that form the basic unit of placement and replication.

As mentioned above, in general Spanner chooses C over A when a partition occurs. In practice, this is due to a few specific choices:

- Use of Paxos groups to achieve consensus on an update; if the leader cannot maintain a quorum due to a partition, updates are stalled and the system is not available (by the CAP definition). Eventually a new leader may emerge, but that also requires a majority.
- Use of two-phase commit for cross-group transactions also means that a partition of the members can prevent commits.

The most likely outcome of a partition in practice is that one side has a quorum and will continue on just fine, perhaps after electing some new leaders. Thus the service continues to be available, but users on the minority side have no access. But this is a case where differential availability matters: those users are likely to have other significant problems, such as no connectivity, and are probably also down. This means that multi-region services built on top of Spanner tend to work relatively well even during a partition. It is possible, but less likely, that some groups will not be available at all.

Transactions in Spanner will work as long as all of the touched groups have a quorum-elected leader and are on one side of the partition. This means that some transactions work perfectly and some will time out, but they are always consistent. An implementation property of Spanner is that any reads that return are consistent, even if the transaction later aborts (for any reason, including time outs).

In addition to normal transactions, Spanner supports snapshot reads, which are read at a particular time in the past. Spanner maintains multiple versions over time, each with a timestamp, and thus can precisely answer snapshot reads with the correct version. In particular, each replica knows the time for which it is caught up (for sure), and any replica can unilaterally answer a read before that time (unless it is way too old and has been garbage collected). Similarly, it is easy to read (asynchronously) at the same time across many groups. Snapshot reads do not need locks at all. In fact, read-only transactions are implemented as a snapshot read at the current time (at any up-to-date replica).

Snapshot reads are thus a little more robust to partitions. In particular, a snapshot read will work if:

1. There is at least one replica for each group on the initiating side of the partition, and
2. The timestamp is in the past for those replicas.

The latter might not be true if the leader is stalled due to a partition, and that could last as long as the partition lasts, since it might not be possible to elect a new leader on this side of the partition. During a partition, it is likely that reads at timestamps prior to the start of the partition will succeed on both sides of the partition, as any reachable replica that has the data suffices.

What about TrueTime?

In general, synchronized clocks can be used to avoid communication in a distributed system. Barbara Liskov provides a fine overview with many examples [Lis91].[3] For our purposes, TrueTime is a global synchronized clock with bounded non-zero error: it returns a time interval that is guaranteed to contain the clock's actual time for some time during the call's execution. Thus, if two intervals do not overlap, then we know calls were definitely ordered in real time. If the intervals overlap, we do not know the actual order.

One subtle thing about Spanner is that it gets serializability from locks, but it gets external consistency (similar to linearizability) from TrueTime. Spanner's external consistency invariant is that for any two transactions, T_1 and T_2 (even if on opposite sides of the globe):

> *if T_2 starts to commit after T_1 finishes committing, then the timestamp for T_2 is greater than the timestamp for T_1.*

Quoting from Liskov [Lis91, section 7]:

> *Synchronized clocks can be used to reduce the probability of having a violation of external consistency. Essentially the primary holds leases, but the object in question is the entire replica group. Each message sent by a backup to the primary gives the primary a lease. The primary can do a read operation unilaterally if it holds unexpired leases from a sub-majority[4] of backups. …*
>
> *The invariant in this system is: whenever a primary performs a read it holds valid leases from a sub-majority of backups. This invariant will not be preserved if clocks get out of synch.*

Spanner's use of TrueTime as the clock ensures the invariant holds. In particular, during a commit, the leader may have to wait until it is sure the commit time is in the past (based on the error bounds). This "commit wait" is not a long wait in practice and it is done in parallel with (internal) transaction communication. In general, external consistency requires monotonically increasing timestamps, and "waiting out the uncertainty" is a common pattern.

Spanner aims to elect leaders for an extended time, typically 10 seconds, by using renewable leases for elected leaders. As discussed by Liskov, every time a quorum agrees on a decision the lease is extended, as the participants just verified that the leadership is effective. When a leader fails there are two options: 1) you can wait for the lease to expire and then elect a new leader, or 2) you can restart the old leader, which might be faster. For some failures, we can send out a "last gasp" UDP packet to release the lease, which is an optimization to speed up expiration. As unplanned failures are rare in a Google data center, the long lease makes sense. The lease also ensures monotonicity of time across leaders, and enables group participants to serve reads within the lease time even without a leader.

[3] There is a direct connection here: two of the Spanner paper authors, Wilson Hsieh and Sanjay Ghemawat, were grad students in Barbara Liskov's extended group in the early 1990s. I was as well.

[4] A "sub-majority" is majority minus one, implying a majority when counting the leader as well.

However, the real value of TrueTime is in what it enables in terms of consistent snapshots. Stepping back a bit, there is a long history of multi-version concurrency-control systems (MVCC) [Ree78] that separately keep old versions and thus allow reading past versions regardless of the current transactional activity. This is a remarkably useful and underrated property: in particular, in Spanner snapshots are consistent (for their time) and thus whatever invariants hold for your system, they will also hold for the snapshot. This is true even when you don't know what the invariants are! Essentially, snapshots are taken in between consecutive transactions and reflect everything up to the time of the snapshot, but nothing more. Without transactionally consistent snapshots, it is difficult to restart from a past time, as the contents may reflect a partially applied transaction that violates some invariant or integrity constraint. It is the lack of consistency that sometimes makes restoring from backups hard to do; in particular, this shows up as some corruption that needs to be fixed by hand.[5]

For example, consider using MapReduce to perform a large analytics query over a database. On Bigtable, which also stores past versions, the notion of time is "jagged" across the data shards, which makes the results unpredictable and sometimes inconsistent (especially for the very recent past). On Spanner, the same MapReduce can pick a precise timestamp and get repeatable and consistent results.

TrueTime also makes it possible to take snapshots across multiple independent systems, as long as they use (monotonically increasing) TrueTime timestamps for commit, agree on a snapshot time, and store multiple versions over time (typically in a log). This is not limited to Spanner: you can make your own transactional system and then ensure snapshots that are consistent across both systems (or even k systems). In general, you need a 2PC (while holding locks) across these systems to agree on the snapshot time and confirm success, but the systems need not agree on anything else, and can be wildly different.

You can also use timestamps as tokens passed through a workflow. For example, if you make an update to a system, you can pass the time of that update to the next stage of the workflow, so that it can tell if its system reflects time after that event. In the case of a partition, this may not be true, in which case the next stage should actually wait if it wants consistency (or proceed if it wants availability). Without the time token, it is hard to know that you need to wait. This isn't the only way to solve this problem, but it does so in a graceful robust way that also ensures eventual consistency. This is particularly useful when the different stages share no code and have different administrators — both can agree on time with no communication.[6]

Snapshots are about the past, but you can also agree on the future. A feature of Spanner is that you can agree on the time in the future for a schema change. This allows you to stage the changes for the new schema so that you are *able* to serve both versions. Once you are ready, you can pick a time to switch to the new schema atomically at all replicas. (You can also pick the time before you stage, but then you might not be ready by the target time.) In theory at least, you can also do future operations, such as a scheduled delete or a change in visibility.

TrueTime itself could be hindered by a partition. The underlying source of time is a combination of GPS receivers and atomic clocks, both of which can maintain accurate time with minuscule drift by themselves. As there are "time masters" in every datacenter (redundantly), it is likely that both sides of a partition would continue to enjoy accurate time. Individual nodes however need network connectivity to the masters, and without it their clocks will drift. Thus, during a partition their intervals slowly grow wider over time, based on bounds on the rate of local clock drift. Operations depending on TrueTime, such as Paxos leader election or transaction commits, thus have to wait a little longer, but the operation still

[5]For comparison, in ARIES [MHL+92], snapshots are intentionally fuzzy, but you can then replay the log on a per-page basis to bring that page up to the target cutoff transaction (and logical time). This works well for recovery, but not so well for running analytics on the snapshot (since it is fuzzy).
[6]Note there is still background communication involved in clock synchronization, including GPS itself and periodic correction traffic.

completes (assuming the 2PC and quorum communication are working).

Conclusion

Spanner reasonably claims to be an "effectively CA" system despite operating over a wide area, as it is always consistent and achieves greater than 5 9s availability. As with Chubby, this combination is possible in practice if you control the whole network, which is rare over the wide area. Even then, it requires significant redundancy of network paths, architectural planning to manage correlated failures, and very careful operations, especially for upgrades. Even then outages will occur, in which case Spanner chooses consistency over availability.

Spanner uses two-phase commit to achieve serializability, but it uses TrueTime for external consistency, consistent reads without locking, and consistent snapshots.

Acknowledgements:
Thanks in particular to Spanner and TrueTime experts Andrew Fikes, Wilson Hsieh, and Peter Hochschild. Additional thanks to Brian Cooper, Kurt Rosenfeld, Chris Taylor, Susan Shepard, Sunil Mushran, Steve Middlekauff, Cliff Frey, Cian Cullinan, Robert Kubis, Deepti Srivastava, Sean Quinlan, Mike Burrows, and Sebastian Kanthak.

References:

[BK14] P. Bailis and K. Kingsbury. The Network is Reliable, *Communications of the ACM*. Vol. 57 No. 9, Pages 48-55. September 2014. Also: https://aphyr.com/posts/288-the-network-is-reliable

[Bre12] E. Brewer. CAP Twelve Years Later: How the "Rules" Have Changed, *IEEE Computer*, Vol. 45, Issue 2, February 2012. pp. 23--29.

[Bur06] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. *Proceedings of OSDI '06: Fourth Symposium on Operating System Design and Implementation*, Seattle, WA, November 2006.

[CDE+12] J. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, JJ Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's Globally-Distributed Database. *Proceedings of OSDI '12: Tenth Symposium on Operating System Design and Implementation*, Hollywood, CA, October, 2012

[Hel16] P. Helland. Standing on Giant Distributed Shoulders: Farsighted Physicists of Yore were Danged Smart! *ACM Queue*, Vol. 14, Issue 2, March-April 2016.

[Lis91] B. Liskov. Practical Uses of Synchronized Clocks in Distributed Systems. *ACM Principles of Distributed Computing (PODC)*. Montreal, Canada, August 1991.

[MHL+92] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh and P. Schwartz. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM Transactions on Database Systems,* Vol. 17, No. 1, March 1992, pp. 94-162.

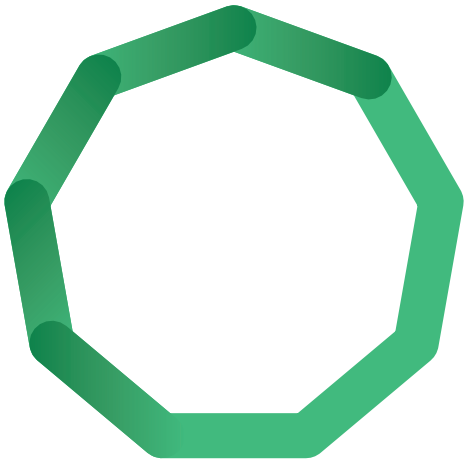
[Ree78] D. Reed. NAMING AND SYNCHRONIZATION IN A DECENTRALIZED COMPUTER SYSTEM, PhD Dissertation, MIT Laboratory for Computer Science, Technical Report MIT-LCS-TR-205. October 1978 [See Section 6.3 for list of versions with timestamps]

Chapter 18

ZooKeeper: Wait-free coordination for Internet-scale systems

Authors:
Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, Benjamin Reed

Publication year: 2010



ZooKeeper: Wait-free coordination for Internet-scale systems

Patrick Hunt and Mahadev Konar
Yahoo! Grid
{phunt,mahadev}@yahoo-inc.com

Flavio P. Junqueira and Benjamin Reed
Yahoo! Research
{fpj,breed}@yahoo-inc.com

Abstract

In this paper, we describe ZooKeeper, a service for coordinating processes of distributed applications. Since ZooKeeper is part of critical infrastructure, ZooKeeper aims to provide a simple and high performance kernel for building more complex coordination primitives at the client. It incorporates elements from group messaging, shared registers, and distributed lock services in a replicated, centralized service. The interface exposed by ZooKeeper has the wait-free aspects of shared registers with an event-driven mechanism similar to cache invalidations of distributed file systems to provide a simple, yet powerful coordination service.

The ZooKeeper interface enables a high-performance service implementation. In addition to the wait-free property, ZooKeeper provides a per client guarantee of FIFO execution of requests and linearizability for all requests that change the ZooKeeper state. These design decisions enable the implementation of a high performance processing pipeline with read requests being satisfied by local servers. We show for the target workloads, 2:1 to 100:1 read to write ratio, that ZooKeeper can handle tens to hundreds of thousands of transactions per second. This performance allows ZooKeeper to be used extensively by client applications.

1 Introduction

Large-scale distributed applications require different forms of coordination. Configuration is one of the most basic forms of coordination. In its simplest form, configuration is just a list of operational parameters for the system processes, whereas more sophisticated systems have dynamic configuration parameters. Group membership and leader election are also common in distributed systems: often processes need to know which other processes are alive and what those processes are in charge of. Locks constitute a powerful coordination primitive that implement mutually exclusive access to critical resources.

One approach to coordination is to develop services for each of the different coordination needs. For example, Amazon Simple Queue Service [3] focuses specifically on queuing. Other services have been developed specifically for leader election [25] and configuration [27]. Services that implement more powerful primitives can be used to implement less powerful ones. For example, Chubby [6] is a locking service with strong synchronization guarantees. Locks can then be used to implement leader election, group membership, etc.

When designing our coordination service, we moved away from implementing specific primitives on the server side, and instead we opted for exposing an API that enables application developers to implement their own primitives. Such a choice led to the implementation of a *coordination kernel* that enables new primitives without requiring changes to the service core. This approach enables multiple forms of coordination adapted to the requirements of applications, instead of constraining developers to a fixed set of primitives.

When designing the API of ZooKeeper, we moved away from blocking primitives, such as locks. Blocking primitives for a coordination service can cause, among other problems, slow or faulty clients to impact negatively the performance of faster clients. The implementation of the service itself becomes more complicated if processing requests depends on responses and failure detection of other clients. Our system, Zookeeper, hence implements an API that manipulates simple *wait-free* data objects organized hierarchically as in file systems. In fact, the ZooKeeper API resembles the one of any other file system, and looking at just the API signatures, ZooKeeper seems to be Chubby without the lock methods, open, and close. Implementing wait-free data objects, however, differentiates ZooKeeper significantly from systems based on blocking primitives such as locks.

Although the wait-free property is important for per-

formance and fault tolerance, it is not sufficient for coordination. We have also to provide order guarantees for operations. In particular, we have found that guaranteeing both *FIFO client ordering* of all operations and *linearizable writes* enables an efficient implementation of the service and it is sufficient to implement coordination primitives of interest to our applications. In fact, we can implement consensus for any number of processes with our API, and according to the hierarchy of Herlihy, ZooKeeper implements a universal object [14].

The ZooKeeper service comprises an ensemble of servers that use replication to achieve high availability and performance. Its high performance enables applications comprising a large number of processes to use such a coordination kernel to manage all aspects of coordination. We were able to implement ZooKeeper using a simple pipelined architecture that allows us to have hundreds or thousands of requests outstanding while still achieving low latency. Such a pipeline naturally enables the execution of operations from a single client in FIFO order. Guaranteeing FIFO client order enables clients to submit operations asynchronously. With asynchronous operations, a client is able to have multiple outstanding operations at a time. This feature is desirable, for example, when a new client becomes a leader and it has to manipulate metadata and update it accordingly. Without the possibility of multiple outstanding operations, the time of initialization can be of the order of seconds instead of sub-second.

To guarantee that update operations satisfy linearizability, we implement a leader-based atomic broadcast protocol [23], called Zab [24]. A typical workload of a ZooKeeper application, however, is dominated by read operations and it becomes desirable to scale read throughput. In ZooKeeper, servers process read operations locally, and we do not use Zab to totally order them.

Caching data on the client side is an important technique to increase the performance of reads. For example, it is useful for a process to cache the identifier of the current leader instead of probing ZooKeeper every time it needs to know the leader. ZooKeeper uses a watch mechanism to enable clients to cache data without managing the client cache directly. With this mechanism, a client can watch for an update to a given data object, and receive a notification upon an update. Chubby manages the client cache directly. It blocks updates to invalidate the caches of all clients caching the data being changed. Under this design, if any of these clients is slow or faulty, the update is delayed. Chubby uses leases to prevent a faulty client from blocking the system indefinitely. Leases, however, only bound the impact of slow or faulty clients, whereas ZooKeeper watches avoid the problem altogether.

In this paper we discuss our design and implementation of ZooKeeper. With ZooKeeper, we are able to implement all coordination primitives that our applications require, even though only writes are linearizable. To validate our approach we show how we implement some coordination primitives with ZooKeeper.

To summarize, in this paper our main contributions are:

Coordination kernel: We propose a wait-free coordination service with relaxed consistency guarantees for use in distributed systems. In particular, we describe our design and implementation of a *coordination kernel*, which we have used in many critical applications to implement various coordination techniques.

Coordination recipes: We show how ZooKeeper can be used to build higher level coordination primitives, even blocking and strongly consistent primitives, that are often used in distributed applications.

Experience with Coordination: We share some of the ways that we use ZooKeeper and evaluate its performance.

2 The ZooKeeper service

Clients submit requests to ZooKeeper through a client API using a ZooKeeper client library. In addition to exposing the ZooKeeper service interface through the client API, the client library also manages the network connections between the client and ZooKeeper servers.

In this section, we first provide a high-level view of the ZooKeeper service. We then discuss the API that clients use to interact with ZooKeeper.

Terminology. In this paper, we use *client* to denote a user of the ZooKeeper service, *server* to denote a process providing the ZooKeeper service, and *znode* to denote an in-memory data node in the ZooKeeper data, which is organized in a hierarchical namespace referred to as the *data tree*. We also use the terms update and write to refer to any operation that modifies the state of the data tree. Clients establish a *session* when they connect to ZooKeeper and obtain a session handle through which they issue requests.

2.1 Service overview

ZooKeeper provides to its clients the abstraction of a set of data nodes (znodes), organized according to a hierarchical name space. The znodes in this hierarchy are data objects that clients manipulate through the ZooKeeper API. Hierarchical name spaces are commonly used in file systems. It is a desirable way of organizing data objects, since users are used to this abstraction and it enables better organization of application meta-data. To refer to a

given znode, we use the standard UNIX notation for file system paths. For example, we use `/A/B/C` to denote the path to znode `C`, where `C` has `B` as its parent and `B` has `A` as its parent. All znodes store data, and all znodes, except for ephemeral znodes, can have children.

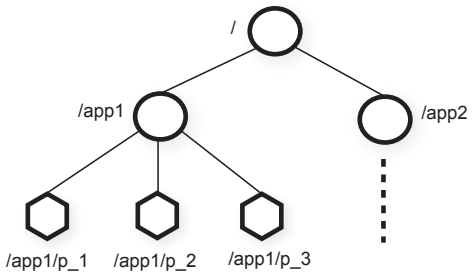


Figure 1: Illustration of ZooKeeper hierarchical name space.

There are two types of znodes that a client can create:

Regular: Clients manipulate regular znodes by creating and deleting them explicitly;

Ephemeral: Clients create such znodes, and they either delete them explicitly, or let the system remove them automatically when the session that creates them terminates (deliberately or due to a failure).

Additionally, when creating a new znode, a client can set a *sequential* flag. Nodes created with the sequential flag set have the value of a monotonically increasing counter appended to its name. If n is the new znode and p is the parent znode, then the sequence value of n is never smaller than the value in the name of any other sequential znode ever created under `p`.

ZooKeeper implements watches to allow clients to receive timely notifications of changes without requiring polling. When a client issues a read operation with a watch flag set, the operation completes as normal except that the server promises to notify the client when the information returned has changed. Watches are one-time triggers associated with a session; they are unregistered once triggered or the session closes. Watches indicate that a change has happened, but do not provide the change. For example, if a client issues a `getData(``/foo'', true)` before "/foo" is changed twice, the client will get one watch event telling the client that data for "/foo" has changed. Session events, such as connection loss events, are also sent to watch callbacks so that clients know that watch events may be delayed.

Data model. The data model of ZooKeeper is essentially a file system with a simplified API and only full data reads and writes, or a key/value table with hierarchical keys. The hierarchal namespace is useful for allocating subtrees for the namespace of different applications and for setting access rights to those subtrees. We also exploit the concept of directories on the client side to build higher level primitives as we will see in section 2.4.

Unlike files in file systems, znodes are not designed for general data storage. Instead, znodes map to abstractions of the client application, typically corresponding to meta-data used for coordination purposes. To illustrate, in Figure 1 we have two subtrees, one for Application 1 (`/app1`) and another for Application 2 (`/app2`). The subtree for Application 1 implements a simple group membership protocol: each client process p_i creates a znode `p_i` under `/app1`, which persists as long as the process is running.

Although znodes have not been designed for general data storage, ZooKeeper does allow clients to store some information that can be used for meta-data or configuration in a distributed computation. For example, in a leader-based application, it is useful for an application server that is just starting to learn which other server is currently the leader. To accomplish this goal, we can have the current leader write this information in a known location in the znode space. Znodes also have associated meta-data with time stamps and version counters, which allow clients to track changes to znodes and execute conditional updates based on the version of the znode.

Sessions. A client connects to ZooKeeper and initiates a session. Sessions have an associated timeout. ZooKeeper considers a client faulty if it does not receive anything from its session for more than that timeout. A session ends when clients explicitly close a session handle or ZooKeeper detects that a clients is faulty. Within a session, a client observes a succession of state changes that reflect the execution of its operations. Sessions enable a client to move transparently from one server to another within a ZooKeeper ensemble, and hence persist across ZooKeeper servers.

2.2 Client API

We present below a relevant subset of the ZooKeeper API, and discuss the semantics of each request.

`create(path, data, flags)`: Creates a znode with path name `path`, stores `data[]` in it, and returns the name of the new znode. `flags` enables a client to select the type of znode: regular, ephemeral, and set the sequential flag;

`delete(path, version)`: Deletes the znode `path` if that znode is at the expected version;

`exists(path, watch)`: Returns true if the znode with path name `path` exists, and returns false otherwise. The `watch` flag enables a client to set a

watch on the znode;

`getData(path, watch)`: Returns the data and meta-data, such as version information, associated with the znode. The `watch` flag works in the same way as it does for `exists()`, except that ZooKeeper does not set the watch if the znode does not exist;

`setData(path, data, version)`: Writes `data[]` to znode `path` if the version number is the current version of the znode;

`getChildren(path, watch)`: Returns the set of names of the children of a znode;

`sync(path)`: Waits for all updates pending at the start of the operation to propagate to the server that the client is connected to. The path is currently ignored.

All methods have both a synchronous and an asynchronous version available through the API. An application uses the synchronous API when it needs to execute a single ZooKeeper operation and it has no concurrent tasks to execute, so it makes the necessary ZooKeeper call and blocks. The asynchronous API, however, enables an application to have both multiple outstanding ZooKeeper operations and other tasks executed in parallel. The ZooKeeper client guarantees that the corresponding callbacks for each operation are invoked in order.

Note that ZooKeeper does not use handles to access znodes. Each request instead includes the full path of the znode being operated on. Not only does this choice simplifies the API (no `open()` or `close()` methods), but it also eliminates extra state that the server would need to maintain.

Each of the update methods take an expected version number, which enables the implementation of conditional updates. If the actual version number of the znode does not match the expected version number the update fails with an unexpected version error. If the version number is -1, it does not perform version checking.

2.3 ZooKeeper guarantees

ZooKeeper has two basic ordering guarantees:

Linearizable writes: all requests that update the state of ZooKeeper are serializable and respect precedence;

FIFO client order: all requests from a given client are executed in the order that they were sent by the client.

Note that our definition of linearizability is different from the one originally proposed by Herlihy [15], and we call it *A-linearizability* (asynchronous linearizability). In the original definition of linearizability by Herlihy, a client is only able to have one outstanding operation at a time (a client is one thread). In ours, we allow a client to have multiple outstanding operations, and consequently we can choose to guarantee no specific order for outstanding operations of the same client or to guarantee FIFO order. We choose the latter for our property. It is important to observe that all results that hold for linearizable objects also hold for A-linearizable objects because a system that satisfies A-linearizability also satisfies linearizability. Because only update requests are A-linearizable, ZooKeeper processes read requests locally at each replica. This allows the service to scale linearly as servers are added to the system.

To see how these two guarantees interact, consider the following scenario. A system comprising a number of processes elects a leader to command worker processes. When a new leader takes charge of the system, it must change a large number of configuration parameters and notify the other processes once it finishes. We then have two important requirements:

- As the new leader starts making changes, we do not want other processes to start using the configuration that is being changed;
- If the new leader dies before the configuration has been fully updated, we do not want the processes to use this partial configuration.

Observe that distributed locks, such as the locks provided by Chubby, would help with the first requirement but are insufficient for the second. With ZooKeeper, the new leader can designate a path as the *ready* znode; other processes will only use the configuration when that znode exists. The new leader makes the configuration change by deleting *ready*, updating the various configuration znodes, and creating *ready*. All of these changes can be pipelined and issued asynchronously to quickly update the configuration state. Although the latency of a change operation is of the order of 2 milliseconds, a new leader that must update 5000 different znodes will take 10 seconds if the requests are issued one after the other; by issuing the requests asynchronously the requests will take less than a second. Because of the ordering guarantees, if a process sees the *ready* znode, it must also see all the configuration changes made by the new leader. If the new leader dies before the *ready* znode is created, the other processes know that the configuration has not been finalized and do not use it.

The above scheme still has a problem: what happens if a process sees that *ready* exists before the new leader starts to make a change and then starts reading the configuration while the change is in progress. This problem is solved by the ordering guarantee for the notifications: if a client is watching for a change, the client will see the notification event before it sees the new state of the system after the change is made. Consequently, if the process that reads the *ready* znode requests to be notified of changes to that znode, it will see a notification inform-

ing the client of the change before it can read any of the new configuration.

Another problem can arise when clients have their own communication channels in addition to ZooKeeper. For example, consider two clients A and B that have a shared configuration in ZooKeeper and communicate through a shared communication channel. If A changes the shared configuration in ZooKeeper and tells B of the change through the shared communication channel, B would expect to see the change when it re-reads the configuration. If B's ZooKeeper replica is slightly behind A's, it may not see the new configuration. Using the above guarantees B can make sure that it sees the most up-to-date information by issuing a write before re-reading the configuration. To handle this scenario more efficiently ZooKeeper provides the `sync` request: when followed by a read, constitutes a *slow read*. `sync` causes a server to apply all pending write requests before processing the read without the overhead of a full write. This primitive is similar in idea to the `flush` primitive of ISIS [5].

ZooKeeper also has the following two liveness and durability guarantees: if a majority of ZooKeeper servers are active and communicating the service will be available; and if the ZooKeeper service responds successfully to a change request, that change persists across any number of failures as long as a quorum of servers is eventually able to recover.

2.4 Examples of primitives

In this section, we show how to use the ZooKeeper API to implement more powerful primitives. The ZooKeeper service knows nothing about these more powerful primitives since they are entirely implemented at the client using the ZooKeeper client API. Some common primitives such as group membership and configuration management are also wait-free. For others, such as rendezvous, clients need to wait for an event. Even though ZooKeeper is wait-free, we can implement efficient blocking primitives with ZooKeeper. ZooKeeper's ordering guarantees allow efficient reasoning about system state, and watches allow for efficient waiting.

Configuration Management ZooKeeper can be used to implement dynamic configuration in a distributed application. In its simplest form configuration is stored in a znode, z_c. Processes start up with the full pathname of z_c. Starting processes obtain their configuration by reading z_c with the watch flag set to true. If the configuration in z_c is ever updated, the processes are notified and read the new configuration, again setting the watch flag to true.

Note that in this scheme, as in most others that use watches, watches are used to make sure that a process has the most recent information. For example, if a process watching z_c is notified of a change to z_c and before it can issue a read for z_c there are three more changes to z_c, the process does not receive three more notification events. This does not affect the behavior of the process, since those three events would have simply notified the process of something it already knows: the information it has for z_c is stale.

Rendezvous Sometimes in distributed systems, it is not always clear a priori what the final system configuration will look like. For example, a client may want to start a master process and several worker processes, but the starting processes is done by a scheduler, so the client does not know ahead of time information such as addresses and ports that it can give the worker processes to connect to the master. We handle this scenario with ZooKeeper using a rendezvous znode, z_r, which is an node created by the client. The client passes the full pathname of z_r as a startup parameter of the master and worker processes. When the master starts it fills in z_r with information about addresses and ports it is using. When workers start, they read z_r with watch set to true. If z_r has not been filled in yet, the worker waits to be notified when z_r is updated. If z_r is an ephemeral node, master and worker processes can watch for z_r to be deleted and clean themselves up when the client ends.

Group Membership We take advantage of ephemeral nodes to implement group membership. Specifically, we use the fact that ephemeral nodes allow us to see the state of the session that created the node. We start by designating a znode, z_g to represent the group. When a process member of the group starts, it creates an ephemeral child znode under z_g. If each process has a unique name or identifier, then that name is used as the name of the child znode; otherwise, the process creates the znode with the SEQUENTIAL flag to obtain a unique name assignment. Processes may put process information in the data of the child znode, addresses and ports used by the process, for example.

After the child znode is created under z_g the process starts normally. It does not need to do anything else. If the process fails or ends, the znode that represents it under z_g is automatically removed.

Processes can obtain group information by simply listing the children of z_g. If a process wants to monitor changes in group membership, the process can set the watch flag to true and refresh the group information (always setting the watch flag to true) when change notifications are received.

Simple Locks Although ZooKeeper is not a lock service, it can be used to implement locks. Applications using ZooKeeper usually use synchronization primitives tailored to their needs, such as those shown above. Here we show how to implement locks with ZooKeeper to show that it can implement a wide variety of general synchronization primitives.

The simplest lock implementation uses "lock files". The lock is represented by a znode. To acquire a lock, a client tries to create the designated znode with the EPHEMERAL flag. If the create succeeds, the client holds the lock. Otherwise, the client can read the znode with the watch flag set to be notified if the current leader dies. A client releases the lock when it dies or explicitly deletes the znode. Other clients that are waiting for a lock try again to acquire a lock once they observe the znode being deleted.

While this simple locking protocol works, it does have some problems. First, it suffers from the herd effect. If there are many clients waiting to acquire a lock, they will all vie for the lock when it is released even though only one client can acquire the lock. Second, it only implements exclusive locking. The following two primitives show how both of these problems can be overcome.

Simple Locks without Herd Effect We define a lock znode l to implement such locks. Intuitively we line up all the clients requesting the lock and each client obtains the lock in order of request arrival. Thus, clients wishing to obtain the lock do the following:

Lock

```
1 n = create(l + "/lock-", EPHEMERAL|SEQUENTIAL)
2 C = getChildren(l, false)
3 if n is lowest znode in C, exit
4 p = znode in C ordered just before n
5 if exists(p, true) wait for watch event
6 goto 2
```

Unlock

```
1 delete(n)
```

The use of the SEQUENTIAL flag in line 1 of `Lock` orders the client's attempt to acquire the lock with respect to all other attempts. If the client's znode has the lowest sequence number at line 3, the client holds the lock. Otherwise, the client waits for deletion of the znode that either has the lock or will receive the lock before this client's znode. By only watching the znode that precedes the client's znode, we avoid the herd effect by only waking up one process when a lock is released or a lock request is abandoned. Once the znode being watched by the client goes away, the client must check if it now holds the lock. (The previous lock request may have been abandoned and there is a znode with a lower sequence number still waiting for or holding the lock.)

Releasing a lock is as simple as deleting the znode n that represents the lock request. By using the EPHEMERAL flag on creation, processes that crash will automatically cleanup any lock requests or release any locks that they may have.

In summary, this locking scheme has the following advantages:

1. The removal of a znode only causes one client to wake up, since each znode is watched by exactly one other client, so we do not have the herd effect;
2. There is no polling or timeouts;
3. Because of the way we have implemented locking, we can see by browsing the ZooKeeper data the amount of lock contention, break locks, and debug locking problems.

Read/Write Locks To implement read/write locks we change the lock procedure slightly and have separate read lock and write lock procedures. The unlock procedure is the same as the global lock case.

Write Lock

```
1  n = create(l + "/write-", EPHEMERAL|SEQUENTIAL)
2  C = getChildren(l, false)
3  if n is lowest znode in C, exit
4  p = znode in C ordered just before n
5  if exists(p, true) wait for event
6  goto 2
```

Read Lock

```
1  n = create(l + "/read-", EPHEMERAL|SEQUENTIAL)
2  C = getChildren(l, false)
3  if no write znodes lower than n in C, exit
4  p = write znode in C ordered just before n
5  if exists(p, true) wait for event
6  goto 3
```

This lock procedure varies slightly from the previous locks. Write locks differ only in naming. Since read locks may be shared, lines 3 and 4 vary slightly because only earlier write lock znodes prevent the client from obtaining a read lock. It may appear that we have a "herd effect" when there are several clients waiting for a read lock and get notified when the "write-" znode with the lower sequence number is deleted; in fact, this is a desired behavior, all those read clients should be released since they may now have the lock.

Double Barrier Double barriers enable clients to synchronize the beginning and the end of a computation. When enough processes, defined by the barrier threshold, have joined the barrier, processes start their computation and leave the barrier once they have finished. We represent a barrier in ZooKeeper with a znode, referred to as b. Every process p registers with b – by creating a znode as a child of b – on entry, and unregisters – removes the child – when it is ready to leave. Processes can enter the barrier when the number of child znodes of b exceeds the barrier threshold. Processes can leave the barrier when all of the processes have removed their children. We use watches to efficiently wait for enter and

exit conditions to be satisfied. To enter, processes watch for the existence of a `ready` child of b that will be created by the process that causes the number of children to exceed the barrier threshold. To leave, processes watch for a particular child to disappear and only check the exit condition once that znode has been removed.

3 ZooKeeper Applications

We now describe some applications that use ZooKeeper, and explain briefly how they use it. We show the primitives of each example in **bold**.

The Fetching Service Crawling is an important part of a search engine, and Yahoo! crawls billions of Web documents. The Fetching Service (FS) is part of the Yahoo! crawler and it is currently in production. Essentially, it has master processes that command page-fetching processes. The master provides the fetchers with configuration, and the fetchers write back informing of their status and health. The main advantages of using ZooKeeper for FS are recovering from failures of masters, guaranteeing availability despite failures, and decoupling the clients from the servers, allowing them to direct their request to healthy servers by just reading their status from ZooKeeper. Thus, FS uses ZooKeeper mainly to manage **configuration metadata**, although it also uses ZooKeeper to elect masters (**leader election**).

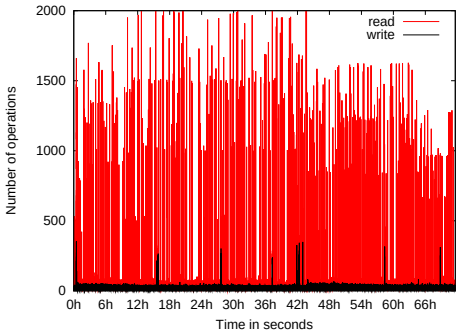


Figure 2: Workload for one ZK server with the Fetching Service. Each point represents a one-second sample.

Figure 2 shows the read and write traffic for a ZooKeeper server used by FS through a period of three days. To generate this graph, we count the number of operations for every second during the period, and each point corresponds to the number of operations in that second. We observe that the read traffic is much higher compared to the write traffic. During periods in which the rate is higher than $1,000$ operations per second, the read:write ratio varies between 10:1 and 100:1. The read operations in this workload are `getData()`, `getChildren()`, and `exists()`, in increasing order of prevalence.

Katta Katta [17] is a distributed indexer that uses ZooKeeper for coordination, and it is an example of a non-Yahoo! application. Katta divides the work of indexing using shards. A master server assigns shards to slaves and tracks progress. Slaves can fail, so the master must redistribute load as slaves come and go. The master can also fail, so other servers must be ready to take over in case of failure. Katta uses ZooKeeper to track the status of slave servers and the master (**group membership**), and to handle master failover (**leader election**). Katta also uses ZooKeeper to track and propagate the assignments of shards to slaves (**configuration management**).

Yahoo! Message Broker Yahoo! Message Broker (YMB) is a distributed publish-subscribe system. The system manages thousands of topics that clients can publish messages to and receive messages from. The topics are distributed among a set of servers to provide scalability. Each topic is replicated using a primary-backup scheme that ensures messages are replicated to two machines to ensure reliable message delivery. The servers that makeup YMB use a shared-nothing distributed architecture which makes coordination essential for correct operation. YMB uses ZooKeeper to manage the distribution of topics (**configuration metadata**), deal with failures of machines in the system (**failure detection** and **group membership**), and control system operation.

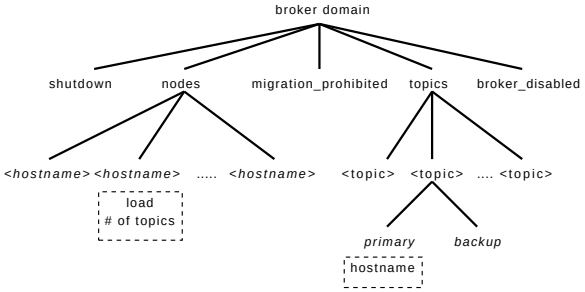


Figure 3: The layout of Yahoo! Message Broker (YMB) structures in ZooKeeper

Figure 3 shows part of the znode data layout for YMB. Each broker domain has a znode called `nodes` that has an ephemeral znode for each of the active servers that compose the YMB service. Each YMB server creates an ephemeral znode under `nodes` with load and status information providing both group membership and status information through ZooKeeper. Nodes such as `shutdown` and `migration_prohibited` are monitored by all of the servers that make up the service and allow centralized control of YMB. The `topics` directory has a child znode for each topic managed by YMB. These topic znodes have child znodes that indicate the

primary and backup server for each topic along with the subscribers of that topic. The `primary` and `backup` server znodes not only allow servers to discover the servers in charge of a topic, but they also manage **leader election** and server crashes.

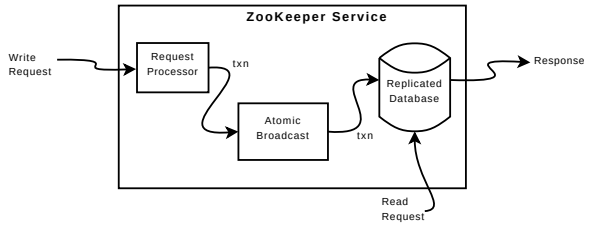


Figure 4: The components of the ZooKeeper service.

4 ZooKeeper Implementation

ZooKeeper provides high availability by replicating the ZooKeeper data on each server that composes the service. We assume that servers fail by crashing, and such faulty servers may later recover. Figure 4 shows the high-level components of the ZooKeeper service. Upon receiving a request, a server prepares it for execution (request processor). If such a request requires coordination among the servers (write requests), then they use an agreement protocol (an implementation of atomic broadcast), and finally servers commit changes to the ZooKeeper database fully replicated across all servers of the ensemble. In the case of read requests, a server simply reads the state of the local database and generates a response to the request.

The replicated database is an *in-memory* database containing the entire data tree. Each znode in the tree stores a maximum of 1MB of data by default, but this maximum value is a configuration parameter that can be changed in specific cases. For recoverability, we efficiently log updates to disk, and we force writes to be on the disk media before they are applied to the in-memory database. In fact, as Chubby [8], we keep a replay log (a write-ahead log, in our case) of committed operations and generate periodic snapshots of the in-memory database.

Every ZooKeeper server services clients. Clients connect to exactly one server to submit its requests. As we noted earlier, read requests are serviced from the local replica of each server database. Requests that change the state of the service, write requests, are processed by an agreement protocol.

As part of the agreement protocol write requests are forwarded to a single server, called the *leader*[1]. The rest of the ZooKeeper servers, called *followers*, receive message proposals consisting of state changes from the leader and agree upon state changes.

4.1 Request Processor

Since the messaging layer is atomic, we guarantee that the local replicas never diverge, although at any point in time some servers may have applied more transactions than others. Unlike the requests sent from clients, the transactions are *idempotent*. When the leader receives a write request, it calculates what the state of the system will be when the write is applied and transforms it into a transaction that captures this new state. The future state must be calculated because there may be outstanding transactions that have not yet been applied to the database. For example, if a client does a conditional `setData` and the version number in the request matches the future version number of the znode being updated, the service generates a `setDataTXN` that contains the new data, the new version number, and updated time stamps. If an error occurs, such as mismatched version numbers or the znode to be updated does not exist, an `errorTXN` is generated instead.

4.2 Atomic Broadcast

All requests that update ZooKeeper state are forwarded to the leader. The leader executes the request and broadcasts the change to the ZooKeeper state through Zab [24], an atomic broadcast protocol. The server that receives the client request responds to the client when it delivers the corresponding state change. Zab uses by default simple majority quorums to decide on a proposal, so Zab and thus ZooKeeper can only work if a majority of servers are correct (*i.e.*, with $2f + 1$ server we can tolerate f failures).

To achieve high throughput, ZooKeeper tries to keep the request processing pipeline full. It may have thousands of requests in different parts of the processing pipeline. Because state changes depend on the application of previous state changes, Zab provides stronger order guarantees than regular atomic broadcast. More specifically, Zab guarantees that changes broadcast by a leader are delivered in the order they were sent and all changes from previous leaders are delivered to an established leader before it broadcasts its own changes.

There are a few implementation details that simplify our implementation and give us excellent performance. We use TCP for our transport so message order is maintained by the network, which allows us to simplify our implementation. We use the leader chosen by Zab as the ZooKeeper leader, so that the same process that creates transactions also proposes them. We use the log to keep track of proposals as the write-ahead log for the in-

[1]Details of leaders and followers, as part of the agreement protocol, are out of the scope of this paper.

memory database, so that we do not have to write messages twice to disk.

During normal operation Zab does deliver all messages in order and exactly once, but since Zab does not persistently record the id of every message delivered, Zab may redeliver a message during recovery. Because we use idempotent transactions, multiple delivery is acceptable as long as they are delivered in order. In fact, ZooKeeper requires Zab to redeliver at least all messages that were delivered after the start of the last snapshot.

4.3 Replicated Database

Each replica has a copy in memory of the ZooKeeper state. When a ZooKeeper server recovers from a crash, it needs to recover this internal state. Replaying all delivered messages to recover state would take prohibitively long after running the server for a while, so ZooKeeper uses periodic snapshots and only requires redelivery of messages since the start of the snapshot. We call ZooKeeper snapshots *fuzzy snapshots* since we do not lock the ZooKeeper state to take the snapshot; instead, we do a depth first scan of the tree atomically reading each znode's data and meta-data and writing them to disk. Since the resulting fuzzy snapshot may have applied some subset of the state changes delivered during the generation of the snapshot, the result may not correspond to the state of ZooKeeper at any point in time. However, since state changes are idempotent, we can apply them twice as long as we apply the state changes in order.

For example, assume that in a ZooKeeper data tree two nodes `/foo` and `/goo` have values `f1` and `g1` respectively and both are at version 1 when the fuzzy snapshot begins, and the following stream of state changes arrive having the form ⟨`transactionType, path, value, new-version`⟩:

```
⟨SetDataTXN, /foo, f2, 2⟩
⟨SetDataTXN, /goo, g2, 2⟩
⟨SetDataTXN, /foo, f3, 3⟩
```

After processing these state changes, `/foo` and `/goo` have values `f3` and `g2` with versions 3 and 2 respectively. However, the fuzzy snapshot may have recorded that `/foo` and `/goo` have values `f3` and `g1` with versions 3 and 1 respectively, which was not a valid state of the ZooKeeper data tree. If the server crashes and recovers with this snapshot and Zab redelivers the state changes, the resulting state corresponds to the state of the service before the crash.

4.4 Client-Server Interactions

When a server processes a write request, it also sends out and clears notifications relative to any watch that corresponds to that update. Servers process writes in order and do not process other writes or reads concurrently. This ensures strict succession of notifications. Note that servers handle notifications locally. Only the server that a client is connected to tracks and triggers notifications for that client.

Read requests are handled locally at each server. Each read request is processed and tagged with a *zxid* that corresponds to the last transaction seen by the server. This *zxid* defines the partial order of the read requests with respect to the write requests. By processing reads locally, we obtain excellent read performance because it is just an in-memory operation on the local server, and there is no disk activity or agreement protocol to run. This design choice is key to achieving our goal of excellent performance with read-dominant workloads.

One drawback of using fast reads is not guaranteeing precedence order for read operations. That is, a read operation may return a stale value, even though a more recent update to the same znode has been committed. Not all of our applications require precedence order, but for applications that do require it, we have implemented `sync`. This primitive executes asynchronously and is ordered by the leader after all pending writes to its local replica. To guarantee that a given read operation returns the latest updated value, a client calls `sync` followed by the read operation. The FIFO order guarantee of client operations together with the global guarantee of `sync` enables the result of the read operation to reflect any changes that happened before the `sync` was issued. In our implementation, we do not need to atomically broadcast `sync` as we use a leader-based algorithm, and we simply place the `sync` operation at the end of the queue of requests between the leader and the server executing the call to `sync`. In order for this to work, the follower must be sure that the leader is still the leader. If there are pending transactions that commit, then the server does not suspect the leader. If the pending queue is empty, the leader needs to issue a null transaction to commit and orders the `sync` after that transaction. This has the nice property that when the leader is under load, no extra broadcast traffic is generated. In our implementation, timeouts are set such that leaders realize they are not leaders before followers abandon them, so we do not issue the null transaction.

ZooKeeper servers process requests from clients in FIFO order. Responses include the *zxid* that the response is relative to. Even heartbeat messages during intervals of no activity include the last *zxid* seen by the server that the client is connected to. If the client connects to a new server, that new server ensures that its view of the ZooKeeper data is at least as recent as the view of the client by checking the last *zxid* of the client against its last *zxid*. If the client has a more recent view than the server, the

server does not reestablish the session with the client until the server has caught up. The client is guaranteed to be able to find another server that has a recent view of the system since the client only sees changes that have been replicated to a majority of the ZooKeeper servers. This behavior is important to guarantee durability.

To detect client session failures, ZooKeeper uses timeouts. The leader determines that there has been a failure if no other server receives anything from a client session within the session timeout. If the client sends requests frequently enough, then there is no need to send any other message. Otherwise, the client sends heartbeat messages during periods of low activity. If the client cannot communicate with a server to send a request or heartbeat, it connects to a different ZooKeeper server to re-establish its session. To prevent the session from timing out, the ZooKeeper client library sends a heartbeat after the session has been idle for $s/3$ ms and switch to a new server if it has not heard from a server for $2s/3$ ms, where s is the session timeout in milliseconds.

5 Evaluation

We performed all of our evaluation on a cluster of 50 servers. Each server has one Xeon dual-core 2.1GHz processor, 4GB of RAM, gigabit ethernet, and two SATA hard drives. We split the following discussion into two parts: throughput and latency of requests.

5.1 Throughput

To evaluate our system, we benchmark throughput when the system is saturated and the changes in throughput for various injected failures. We varied the number of servers that make up the ZooKeeper service, but always kept the number of clients the same. To simulate a large number of clients, we used 35 machines to simulate 250 simultaneous clients.

We have a Java implementation of the ZooKeeper server, and both Java and C clients[2]. For these experiments, we used the Java server configured to log to one dedicated disk and take snapshots on another. Our benchmark client uses the asynchronous Java client API, and each client has at least 100 requests outstanding. Each request consists of a read or write of 1K of data. We do not show benchmarks for other operations since the performance of all the operations that modify state are approximately the same, and the performance of non-state modifying operations, excluding `sync`, are approximately the same. (The performance of `sync` approximates that of a light-weight write, since the request must go to the leader, but does not get broadcast.) Clients send counts of the number of completed operations every $300ms$ and we sample every $6s$. To prevent memory overflows, servers throttle the number of concurrent requests in the system. ZooKeeper uses request throttling to keep servers from being overwhelmed. For these experiments, we configured the ZooKeeper servers to have a maximum of $2,000$ total requests in process.

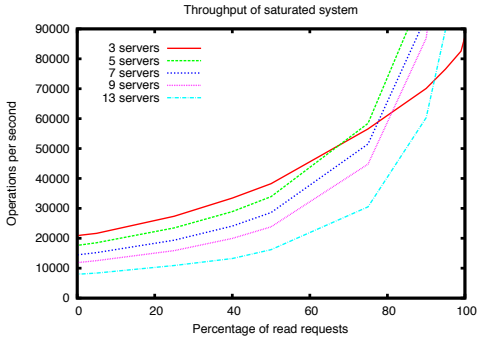


Figure 5: The throughput performance of a saturated system as the ratio of reads to writes vary.

Servers	100% **Reads**	0% **Reads**
13	460k	8k
9	296k	12k
7	257k	14k
5	165k	18k
3	87k	21k

Table 1: The throughput performance of the extremes of a saturated system.

In Figure 5, we show throughput as we vary the ratio of read to write requests, and each curve corresponds to a different number of servers providing the ZooKeeper service. Table 1 shows the numbers at the extremes of the read loads. Read throughput is higher than write throughput because reads do not use atomic broadcast. The graph also shows that the number of servers also has a negative impact on the performance of the broadcast protocol. From these graphs, we observe that the number of servers in the system does not only impact the number of failures that the service can handle, but also the workload the service can handle. Note that the curve for three servers crosses the others around 60%. This situation is not exclusive of the three-server configuration, and happens for all configurations due to the parallelism local reads enable. It is not observable for other configurations in the figure, however, because we have capped the maximum y-axis throughput for readability.

There are two reasons for write requests taking longer than read requests. First, write requests must go through atomic broadcast, which requires some extra processing

[2]The implementation is publicly available at `http://hadoop.apache.org/zookeeper`.

and adds latency to requests. The other reason for longer processing of write requests is that servers must ensure that transactions are logged to non-volatile store before sending acknowledgments back to the leader. In principle, this requirement is excessive, but for our production systems we trade performance for reliability since ZooKeeper constitutes application ground truth. We use more servers to tolerate more faults. We increase write throughput by partitioning the ZooKeeper data into multiple ZooKeeper ensembles. This performance trade off between replication and partitioning has been previously observed by Gray *et al.* [12].

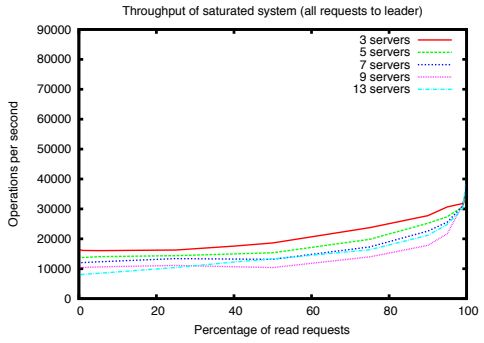


Figure 6: Throughput of a saturated system, varying the ratio of reads to writes when all clients connect to the leader.

ZooKeeper is able to achieve such high throughput by distributing load across the servers that makeup the service. We can distribute the load because of our relaxed consistency guarantees. Chubby clients instead direct all requests to the leader. Figure 6 shows what happens if we do not take advantage of this relaxation and forced the clients to only connect to the leader. As expected the throughput is much lower for read-dominant workloads, but even for write-dominant workloads the throughput is lower. The extra CPU and network load caused by servicing clients impacts the ability of the leader to coordinate the broadcast of the proposals, which in turn adversely impacts the overall write performance.

The atomic broadcast protocol does most of the work of the system and thus limits the performance of ZooKeeper more than any other component. Figure 7 shows the throughput of the atomic broadcast component. To benchmark its performance we simulate clients by generating the transactions directly at the leader, so there is no client connections or client requests and replies. At maximum throughput the atomic broadcast component becomes CPU bound. In theory the performance of Figure 7 would match the performance of ZooKeeper with 100% writes. However, the ZooKeeper client communication, ACL checks, and request to transaction con-

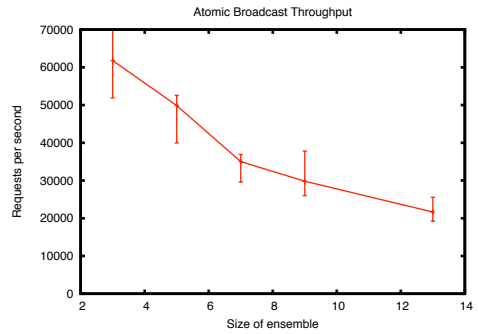


Figure 7: Average throughput of the atomic broadcast component in isolation. Error bars denote the minimum and maximum values.

versions all require CPU. The contention for CPU lowers ZooKeeper throughput to substantially less than the atomic broadcast component in isolation. Because ZooKeeper is a critical production component, up to now our development focus for ZooKeeper has been correctness and robustness. There are plenty of opportunities for improving performance significantly by eliminating things like extra copies, multiple serializations of the same object, more efficient internal data structures, etc.

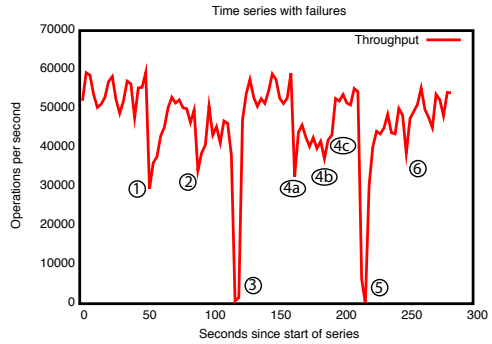


Figure 8: Throughput upon failures.

To show the behavior of the system over time as failures are injected we ran a ZooKeeper service made up of 5 machines. We ran the same saturation benchmark as before, but this time we kept the write percentage at a constant 30%, which is a conservative ratio of our expected workloads. Periodically we killed some of the server processes. Figure 8 shows the system throughput as it changes over time. The events marked in the figure are the following:

1. Failure and recovery of a follower;
2. Failure and recovery of a different follower;
3. Failure of the leader;
4. Failure of two followers (a, b) in the first two marks, and recovery at the third mark (c);
5. Failure of the leader.

6. Recovery of the leader.

There are a few important observations from this graph. First, if followers fail and recover quickly, then ZooKeeper is able to sustain a high throughput despite the failure. The failure of a single follower does not prevent servers from forming a quorum, and only reduces throughput roughly by the share of read requests that the server was processing before failing. Second, our leader election algorithm is able to recover fast enough to prevent throughput from dropping substantially. In our observations, ZooKeeper takes less than $200ms$ to elect a new leader. Thus, although servers stop serving requests for a fraction of second, we do not observe a throughput of zero due to our sampling period, which is on the order of seconds. Third, even if followers take more time to recover, ZooKeeper is able to raise throughput again once they start processing requests. One reason that we do not recover to the full throughput level after events 1, 2, and 4 is that the clients only switch followers when their connection to the follower is broken. Thus, after event 4 the clients do not redistribute themselves until the leader fails at events 3 and 5. In practice such imbalances work themselves out over time as clients come and go.

5.2 Latency of requests

To assess the latency of requests, we created a benchmark modeled after the Chubby benchmark [6]. We create a worker process that simply sends a create, waits for it to finish, sends an asynchronous delete of the new node, and then starts the next create. We vary the number of workers accordingly, and for each run, we have each worker create 50,000 nodes. We calculate the throughput by dividing the number of create requests completed by the total time it took for all the workers to complete.

	Number of servers			
Workers	3	5	7	9
1	776	748	758	711
10	2074	1832	1572	1540
20	2740	2336	1934	1890

Table 2: Create requests processed per second.

Table 2 show the results of our benchmark. The create requests include 1K of data, rather than 5 bytes in the Chubby benchmark, to better coincide with our expected use. Even with these larger requests, the throughput of ZooKeeper is more than 3 times higher than the published throughput of Chubby. The throughput of the single ZooKeeper worker benchmark indicates that the average request latency is 1.2ms for three servers and 1.4ms for 9 servers.

	# of clients		
# of barriers	**50**	**100**	**200**
200	9.4	19.8	41.0
400	16.4	34.1	62.0
800	28.9	55.9	112.1
1600	54.0	102.7	234.4

Table 3: Barrier experiment with time in seconds. Each point is the average of the time for each client to finish over five runs.

5.3 Performance of barriers

In this experiment, we execute a number of barriers sequentially to assess the performance of primitives implemented with ZooKeeper. For a given number of barriers b, each client first enters all b barriers, and then it leaves all b barriers in succession. As we use the double-barrier algorithm of Section 2.4, a client first waits for all other clients to execute the `enter()` procedure before moving to next call (similarly for `leave()`).

We report the results of our experiments in Table 3. In this experiment, we have 50, 100, and 200 clients entering a number b of barriers in succession, $b \in \{200, 400, 800, 1600\}$. Although an application can have thousands of ZooKeeper clients, quite often a much smaller subset participates in each coordination operation as clients are often grouped according to the specifics of the application.

Two interesting observations from this experiment are that the time to process all barriers increase roughly linearly with the number of barriers, showing that concurrent access to the same part of the data tree did not produce any unexpected delay, and that latency increases proportionally to the number of clients. This is a consequence of not saturating the ZooKeeper service. In fact, we observe that even with clients proceeding in lock-step, the throughput of barrier operations (enter and leave) is between 1,950 and 3,100 operations per second in all cases. In ZooKeeper operations, this corresponds to throughput values between 10,700 and 17,000 operations per second. As in our implementation we have a ratio of reads to writes of 4:1 (80% of read operations), the throughput our benchmark code uses is much lower compared to the raw throughput ZooKeeper can achieve (over 40,000 according to Figure 5). This is due to clients waiting on other clients.

6 Related work

ZooKeeper has the goal of providing a service that mitigates the problem of coordinating processes in distributed applications. To achieve this goal, its design uses ideas from previous coordination services, fault tolerant systems, distributed algorithms, and file systems.

We are not the first to propose a system for the coordination of distributed applications. Some early systems propose a distributed lock service for transactional applications [13], and for sharing information in clusters of computers [19]. More recently, Chubby proposes a system to manage advisory locks for distributed applications [6]. Chubby shares several of the goals of ZooKeeper. It also has a file-system-like interface, and it uses an agreement protocol to guarantee the consistency of the replicas. However, ZooKeeper is not a lock service. It can be used by clients to implement locks, but there are no lock operations in its API. Unlike Chubby, ZooKeeper allows clients to connect to any ZooKeeper server, not just the leader. ZooKeeper clients can use their local replicas to serve data and manage watches since its consistency model is much more relaxed than Chubby. This enables ZooKeeper to provide higher performance than Chubby, allowing applications to make more extensive use of ZooKeeper.

There have been fault-tolerant systems proposed in the literature with the goal of mitigating the problem of building fault-tolerant distributed applications. One early system is ISIS [5]. The ISIS system transforms abstract type specifications into fault-tolerant distributed objects, thus making fault-tolerance mechanisms transparent to users. Horus [30] and Ensemble [31] are systems that evolved from ISIS. ZooKeeper embraces the notion of virtual synchrony of ISIS. Finally, Totem guarantees total order of message delivery in an architecture that exploits hardware broadcasts of local area networks [22]. ZooKeeper works with a wide variety of network topologies which motivated us to rely on TCP connections between server processes and not assume any special topology or hardware features. We also do not expose any of the ensemble communication used internally in ZooKeeper.

One important technique for building fault-tolerant services is state-machine replication [26], and Paxos [20] is an algorithm that enables efficient implementations of replicated state-machines for asynchronous systems. We use an algorithm that shares some of the characteristics of Paxos, but that combines transaction logging needed for consensus with write-ahead logging needed for data tree recovery to enable an efficient implementation. There have been proposals of protocols for practical implementations of Byzantine-tolerant replicated state-machines [7, 10, 18, 1, 28]. ZooKeeper does not assume that servers can be Byzantine, but we do employ mechanisms such as checksums and sanity checks to catch non-malicious Byzantine faults. Clement *et al.* discuss an approach to make ZooKeeper fully Byzantine fault-tolerant without modifying the current server code base [9]. To date, we have not observed faults in production that would have been prevented using a fully Byzantine fault-tolerant protocol. [29].

Boxwood [21] is a system that uses distributed lock servers. Boxwood provides higher-level abstractions to applications, and it relies upon a distributed lock service based on Paxos. Like Boxwood, ZooKeeper is a component used to build distributed systems. ZooKeeper, however, has high-performance requirements and is used more extensively in client applications. ZooKeeper exposes lower-level primitives that applications use to implement higher-level primitives.

ZooKeeper resembles a small file system, but it only provides a small subset of the file system operations and adds functionality not present in most file systems such as ordering guarantees and conditional writes. ZooKeeper watches, however, are similar in spirit to the cache callbacks of AFS [16].

Sinfonia [2] introduces *mini-transactions*, a new paradigm for building scalable distributed systems. Sinfonia has been designed to store application data, whereas ZooKeeper stores application metadata. ZooKeeper keeps its state fully replicated and in memory for high performance and consistent latency. Our use of file system like operations and ordering enables functionality similar to mini-transactions. The znode is a convenient abstraction upon which we add watches, a functionality missing in Sinfonia. Dynamo [11] allows clients to get and put relatively small (less than 1M) amounts of data in a distributed key-value store. Unlike ZooKeeper, the key space in Dynamo is not hierarchal. Dynamo also does not provide strong durability and consistency guarantees for writes, but instead resolves conflicts on reads.

DepSpace [4] uses a tuple space to provide a Byzantine fault-tolerant service. Like ZooKeeper DepSpace uses a simple server interface to implement strong synchronization primitives at the client. While DepSpace's performance is much lower than ZooKeeper, it provides stronger fault tolerance and confidentiality guarantees.

7 Conclusions

ZooKeeper takes a wait-free approach to the problem of coordinating processes in distributed systems, by exposing wait-free objects to clients. We have found ZooKeeper to be useful for several applications inside and outside Yahoo!. ZooKeeper achieves throughput values of hundreds of thousands of operations per second for read-dominant workloads by using fast reads with watches, both of which served by local replicas. Although our consistency guarantees for reads and watches appear to be weak, we have shown with our use cases that this combination allows us to implement efficient and sophisticated coordination protocols at the client even though reads are not precedence-ordered and the implementation of data objects is wait-free. The wait-free property has proved to be essential for high performance.

Although we have described only a few applications, there are many others using ZooKeeper. We believe such a success is due to its simple interface and the powerful abstractions that one can implement through this interface. Further, because of the high-throughput of ZooKeeper, applications can make extensive use of it, not only course-grained locking.

Acknowledgements

We would like to thank Andrew Kornev and Runping Qi for their contributions to ZooKeeper; Zeke Huang and Mark Marchukov for valuable feedback; Brian Cooper and Laurence Ramontianu for their early contributions to ZooKeeper; Brian Bershad and Geoff Voelker made important comments on the presentation.

References

[1] M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. K. Reiter, and J. J. Wylie. Fault-scalable byzantine fault-tolerant services. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 59–74, New York, NY, USA, 2005. ACM.

[2] M. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: A new paradigm for building scalable distributed systems. In *SOSP '07: Proceedings of the 21st ACM symposium on Operating systems principles*, New York, NY, 2007.

[3] Amazon. Amazon simple queue service. `http://aws.amazon.com/sqs/`, 2008.

[4] A. N. Bessani, E. P. Alchieri, M. Correia, and J. da Silva Fraga. Depspace: A byzantine fault-tolerant coordination service. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Systems Conference - EuroSys 2008*, Apr. 2008.

[5] K. P. Birman. Replication and fault-tolerance in the ISIS system. In *SOSP '85: Proceedings of the 10th ACM symposium on Operating systems principles*, New York, USA, 1985. ACM Press.

[6] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th ACM/USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[7] M. Castro and B. Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20(4), 2002.

[8] T. Chandra, R. Griesemer, and J. Redstone. Paxos made live: An engineering perspective. In *Proceedings of the 26th annual ACM symposium on Principles of distributed computing (PODC)*, Aug. 2007.

[9] A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. Riche. UpRight cluster services. In *Proceedings of the 22 nd ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2009.

[10] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shira. Hq replication: A hybrid quorum protocol for byzantine fault tolerance. In *SOSP '07: Proceedings of the 21st ACM symposium on Operating systems principles*, New York, NY, USA, 2007.

[11] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazons highly available key-value store. In *SOSP '07: Proceedings of the 21st ACM symposium on Operating systems principles*, New York, NY, USA, 2007. ACM Press.

[12] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *Proceedings of SIGMOD '96*, pages 173–182, New York, NY, USA, 1996. ACM.

[13] A. Hastings. Distributed lock management in a transaction processing environment. In *Proceedings of IEEE 9th Symposium on Reliable Distributed Systems*, Oct. 1990.

[14] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1), 1991.

[15] M. Herlihy and J. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3), July 1990.

[16] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. *ACM Trans. Comput. Syst.*, 6(1), 1988.

[17] Katta. Katta - distribute lucene indexes in a grid. `http://katta.wiki.sourceforge.net/`, 2008.

[18] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: speculative byzantine fault tolerance. *SIGOPS Oper. Syst. Rev.*, 41(6):45–58, 2007.

[19] N. P. Kronenberg, H. M. Levy, and W. D. Strecker. Vaxclusters (extended abstract): a closely-coupled distributed system. *SIGOPS Oper. Syst. Rev.*, 19(5), 1985.

[20] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2), May 1998.

[21] J. MacCormick, N. Murphy, M. Najork, C. A. Thekkath, and L. Zhou. Boxwood: Abstractions as the foundation for storage infrastructure. In *Proceedings of the 6th ACM/USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

[22] L. Moser, P. Melliar-Smith, D. Agarwal, R. Budhia, C. Lingley-Papadopoulos, and T. Archambault. The totem system. In *Proceedings of the 25th International Symposium on Fault-Tolerant Computing*, June 1995.

[23] S. Mullender, editor. *Distributed Systems, 2nd edition*. ACM Press, New York, NY, USA, 1993.

[24] B. Reed and F. P. Junqueira. A simple totally ordered broadcast protocol. In *LADIS '08: Proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware*, pages 1–6, New York, NY, USA, 2008. ACM.

[25] N. Schiper and S. Toueg. A robust and lightweight stable leader election service for dynamic systems. In *DSN*, 2008.

[26] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4), 1990.

[27] A. Sherman, P. A. Lisiecki, A. Berkheimer, and J. Wein. ACMS: The Akamai configuration management system. In *NSDI*, 2005.

[28] A. Singh, P. Fonseca, P. Kuznetsov, R. Rodrigues, and P. Maniatis. Zeno: eventually consistent byzantine-fault tolerance. In *NSDI'09: Proceedings of the 6th USENIX symposium on Networked systems design and implementation*, pages 169–184, Berkeley, CA, USA, 2009. USENIX Association.

[29] Y. J. Song, F. Junqueira, and B. Reed. BFT for the skeptics. `http://www.net.t-labs.tu-berlin.de/~petr/BFTW3/abstracts/talk-abstract.pdf`.

[30] R. van Renesse and K. Birman. Horus, a flexible group communication systems. *Communications of the ACM*, 39(16), Apr. 1996.

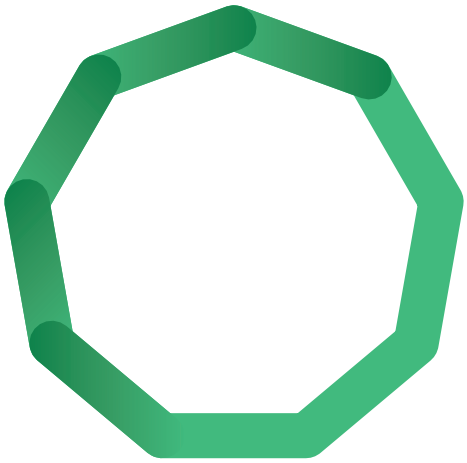
[31] R. van Renesse, K. Birman, M. Hayden, A. Vaysburd, and D. Karr. Building adaptive systems using ensemble. *Software - Practice and Experience*, 28(5), July 1998.

Chapter 19

Kafka: a Distributed Messaging System for Log Processing

Authors:
Jay Kreps, Neha Narkhede, Jun Rao

Publication year: 2011



Kafka: a Distributed Messaging System for Log Processing

Jay Kreps
LinkedIn Corp.
jkreps@linkedin.com

Neha Narkhede
LinkedIn Corp.
nnarkhede@linkedin.com

Jun Rao
LinkedIn Corp.
jrao@linkedin.com

ABSTRACT
Log processing has become a critical component of the data pipeline for consumer internet companies. We introduce Kafka, a distributed messaging system that we developed for collecting and delivering high volumes of log data with low latency. Our system incorporates ideas from existing log aggregators and messaging systems, and is suitable for both offline and online message consumption. We made quite a few unconventional yet practical design choices in Kafka to make our system efficient and scalable. Our experimental results show that Kafka has superior performance when compared to two popular messaging systems. We have been using Kafka in production for some time and it is processing hundreds of gigabytes of new data each day.

General Terms
Management, Performance, Design, Experimentation.

Keywords
messaging, distributed, log processing, throughput, online.

1. Introduction
There is a large amount of "log" data generated at any sizable internet company. This data typically includes (1) user activity events corresponding to logins, pageviews, clicks, "likes", sharing, comments, and search queries; (2) operational metrics such as service call stack, call latency, errors, and system metrics such as CPU, memory, network, or disk utilization on each machine. Log data has long been a component of analytics used to track user engagement, system utilization, and other metrics. However recent trends in internet applications have made activity data a part of the production data pipeline used directly in site features. These uses include (1) search relevance, (2) recommendations which may be driven by item popularity or co-occurrence in the activity stream, (3) ad targeting and reporting, and (4) security applications that protect against abusive behaviors such as spam or unauthorized data scraping, and (5) newsfeed features that aggregate user status updates or actions for their "friends" or "connections" to read.

This production, real-time usage of log data creates new challenges for data systems because its volume is orders of magnitude larger than the "real" data. For example, search, recommendations, and advertising often require computing granular click-through rates, which generate log records not only for every user click, but also for dozens of items on each page that are not clicked. Every day, China Mobile collects 5–8TB of phone call records [11] and Facebook gathers almost 6TB of various user activity events [12].

Many early systems for processing this kind of data relied on physically scraping log files off production servers for analysis. In recent years, several specialized distributed log aggregators have been built, including Facebook's Scribe [6], Yahoo's Data Highway [4], and Cloudera's Flume [3]. Those systems are primarily designed for collecting and loading the log data into a data warehouse or Hadoop [8] for offline consumption. At LinkedIn (a social network site), we found that in addition to traditional offline analytics, we needed to support most of the real-time applications mentioned above with delays of no more than a few seconds.

We have built a novel messaging system for log processing called Kafka [18] that combines the benefits of traditional log aggregators and messaging systems. On the one hand, Kafka is distributed and scalable, and offers high throughput. On the other hand, Kafka provides an API similar to a messaging system and allows applications to consume log events in real time. Kafka has been open sourced and used successfully in production at LinkedIn for more than 6 months. It greatly simplifies our infrastructure, since we can exploit a single piece of software for both online and offline consumption of the log data of all types. The rest of the paper is organized as follows. We revisit traditional messaging systems and log aggregators in Section 2. In Section 3, we describe the architecture of Kafka and its key design principles. We describe our deployment of Kafka at LinkedIn in Section 4 and the performance results of Kafka in Section 5. We discuss future work and conclude in Section 6.

2. Related Work
Traditional enterprise messaging systems [1][7][15][17] have existed for a long time and often play a critical role as an event bus for processing asynchronous data flows. However, there are a few reasons why they tend not to be a good fit for log processing. First, there is a mismatch in features offered by enterprise systems. Those systems often focus on offering a rich set of delivery guarantees. For example, IBM Websphere MQ [7] has transactional supports that allow an application to insert messages into multiple queues atomically. The JMS [14] specification allows each individual message to be acknowledged after consumption, potentially out of order. Such delivery guarantees are often overkill for collecting log data. For instance, losing a few pageview events occasionally is certainly not the end of the world. Those unneeded features tend to increase the complexity of both the API and the underlying implementation of those systems. Second, many systems do not focus as strongly on throughput as their primary design constraint. For example, JMS has no API to allow the producer to explicitly batch multiple messages into a

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
NetDB'11, Jun. 12, 2011, Athens, Greece.
Copyright 2011 ACM 978-1-4503-0652-2/11/06…$10.00.

single request. This means each message requires a full TCP/IP roundtrip, which is not feasible for the throughput requirements of our domain. Third, those systems are weak in distributed support. There is no easy way to partition and store messages on multiple machines. Finally, many messaging systems assume near immediate consumption of messages, so the queue of unconsumed messages is always fairly small. Their performance degrades significantly if messages are allowed to accumulate, as is the case for offline consumers such as data warehousing applications that do periodic large loads rather than continuous consumption.

A number of specialized log aggregators have been built over the last few years. Facebook uses a system called Scribe. Each front-end machine can send log data to a set of Scribe machines over sockets. Each Scribe machine aggregates the log entries and periodically dumps them to HDFS [9] or an NFS device. Yahoo's data highway project has a similar dataflow. A set of machines aggregate events from the clients and roll out "minute" files, which are then added to HDFS. Flume is a relatively new log aggregator developed by Cloudera. It supports extensible "pipes" and "sinks", and makes streaming log data very flexible. It also has more integrated distributed support. However, most of those systems are built for consuming the log data offline, and often expose implementation details unnecessarily (e.g. "minute files") to the consumer. Additionally, most of them use a "push" model in which the broker forwards data to consumers. At LinkedIn, we find the "pull" model more suitable for our applications since each consumer can retrieve the messages at the maximum rate it can sustain and avoid being flooded by messages pushed faster than it can handle. The pull model also makes it easy to rewind a consumer and we discuss this benefit at the end of Section 3.2.

More recently, Yahoo! Research developed a new distributed pub/sub system called HedWig [13]. HedWig is highly scalable and available, and offers strong durability guarantees. However, it is mainly intended for storing the commit log of a data store.

3. Kafka Architecture and Design Principles

Because of limitations in existing systems, we developed a new messaging-based log aggregator Kafka. We first introduce the basic concepts in Kafka. A stream of messages of a particular type is defined by a *topic*. A producer can publish messages to a topic. The published messages are then stored at a set of servers called *brokers*. A consumer can subscribe to one or more topics from the brokers, and consume the subscribed messages by pulling data from the brokers.

Messaging is conceptually simple, and we have tried to make the Kafka API equally simple to reflect this. Instead of showing the exact API, we present some sample code to show how the API is used. The sample code of the producer is given below. A message is defined to contain just a payload of bytes. A user can choose her favorite serialization method to encode a message. For efficiency, the producer can send a set of messages in a single publish request.

Sample producer code:

```
producer = new Producer(…);
message = new Message("test message str".getBytes());
set = new MessageSet(message);
producer.send("topic1", set);
```

To subscribe to a topic, a consumer first creates one or more message streams for the topic. The messages published to that topic will be evenly distributed into these sub-streams. The details about how Kafka distributes the messages are described later in Section 3.2. Each message stream provides an iterator interface over the continual stream of messages being produced. The consumer then iterates over every message in the stream and processes the payload of the message. Unlike traditional iterators, the message stream iterator never terminates. If there are currently no more messages to consume, the iterator blocks until new messages are published to the topic. We support both the point-to-point delivery model in which multiple consumers jointly consume a single copy of all messages in a topic, as well as the publish/subscribe model in which multiple consumers each retrieve its own copy of a topic.

Sample consumer code:

```
streams[] = Consumer.createMessageStreams("topic1", 1)
for (message : streams[0]) {
  bytes = message.payload();
  // do something with the bytes
}
```

The overall architecture of Kafka is shown in Figure 1. Since Kafka is distributed in nature, an Kafka cluster typically consists of multiple brokers. To balance load, a topic is divided into multiple partitions and each broker stores one or more of those partitions. Multiple producers and consumers can publish and retrieve messages at the same time. In Section 3.1, we describe the layout of a single partition on a broker and a few design choices that we selected to make accessing a partition efficient. In Section 3.2, we describe how the producer and the consumer interact with multiple brokers in a distributed setting. We discuss the delivery guarantees of Kafka in Section 3.3.

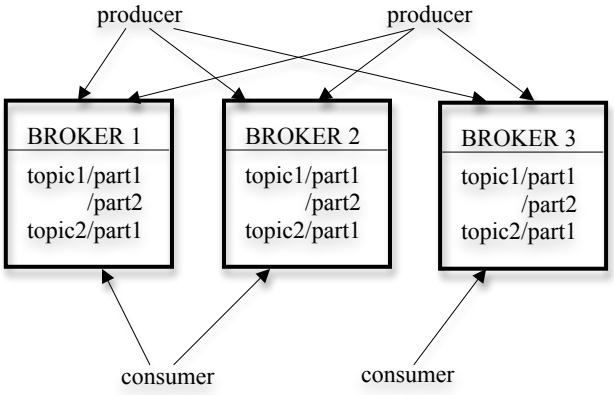


Figure 1. Kafka Architecture

3.1 Efficiency on a Single Partition

We made a few decisions in Kafka to make the system efficient.

Simple storage: Kafka has a very simple storage layout. Each partition of a topic corresponds to a logical log. Physically, a log is implemented as a set of segment files of approximately the same size (e.g., 1GB). Every time a producer publishes a message to a partition, the broker simply appends the message to the last segment file. For better performance, we flush the segment files to disk only after a configurable number of messages have been published or a certain amount of time has elapsed. A message is only exposed to the consumers after it is flushed.

Unlike typical messaging systems, a message stored in Kafka doesn't have an explicit message id. Instead, each message is addressed by its logical offset in the log. This avoids the overhead of maintaining auxiliary, seek-intensive random-access index structures that map the message ids to the actual message locations. Note that our message ids are increasing but not consecutive. To compute the id of the next message, we have to add the length of the current message to its id. From now on, we will use message ids and offsets interchangeably.

A consumer always consumes messages from a particular partition sequentially. If the consumer acknowledges a particular message offset, it implies that the consumer has received all messages prior to that offset in the partition. Under the covers, the consumer is issuing asynchronous pull requests to the broker to have a buffer of data ready for the application to consume. Each pull request contains the offset of the message from which the consumption begins and an acceptable number of bytes to fetch. Each broker keeps in memory a sorted list of offsets, including the offset of the first message in every segment file. The broker locates the segment file where the requested message resides by searching the offset list, and sends the data back to the consumer. After a consumer receives a message, it computes the offset of the next message to consume and uses it in the next pull request. The layout of an Kafka log and the in-memory index is depicted in Figure 2. Each box shows the offset of a message.

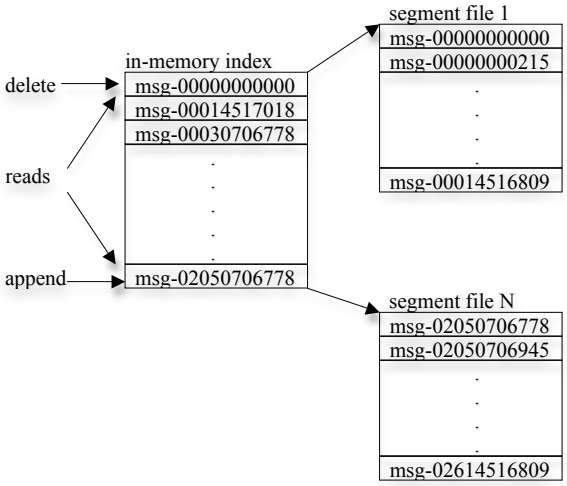


Figure 2. Kafka log

Efficient transfer: We are very careful about transferring data in and out of Kafka. Earlier, we have shown that the producer can submit a set of messages in a single send request. Although the end consumer API iterates one message at a time, under the covers, each pull request from a consumer also retrieves multiple messages up to a certain size, typically hundreds of kilobytes.

Another unconventional choice that we made is to avoid explicitly caching messages in memory at the Kafka layer. Instead, we rely on the underlying file system page cache. This has the main benefit of avoiding double buffering---messages are only cached in the page cache. This has the additional benefit of retaining warm cache even when a broker process is restarted. Since Kafka doesn't cache messages in process at all, it has very little overhead in garbage collecting its memory, making efficient implementation in a VM-based language feasible. Finally, since both the producer and the consumer access the segment files sequentially, with the consumer often lagging the producer by a small amount, normal operating system caching heuristics are very effective (specifically write-through caching and read-ahead). We have found that both the production and the consumption have consistent performance linear to the data size, up to many terabytes of data.

In addition we optimize the network access for consumers. Kafka is a multi-subscriber system and a single message may be consumed multiple times by different consumer applications. A typical approach to sending bytes from a local file to a remote socket involves the following steps: (1) read data from the storage media to the page cache in an OS, (2) copy data in the page cache to an application buffer, (3) copy application buffer to another kernel buffer, (4) send the kernel buffer to the socket. This includes 4 data copying and 2 system calls. On Linux and other Unix operating systems, there exists a sendfile API [5] that can directly transfer bytes from a file channel to a socket channel. This typically avoids 2 of the copies and 1 system call introduced in steps (2) and (3). Kafka exploits the sendfile API to efficiently deliver bytes in a log segment file from a broker to a consumer.

Stateless broker: Unlike most other messaging systems, in Kafka, the information about how much each consumer has consumed is not maintained by the broker, but by the consumer itself. Such a design reduces a lot of the complexity and the overhead on the broker. However, this makes it tricky to delete a message, since a broker doesn't know whether all subscribers have consumed the message. Kafka solves this problem by using a simple time-based SLA for the retention policy. A message is automatically deleted if it has been retained in the broker longer than a certain period, typically 7 days. This solution works well in practice. Most consumers, including the offline ones, finish consuming either daily, hourly, or in real-time. The fact that the performance of Kafka doesn't degrade with a larger data size makes this long retention feasible.

There is an important side benefit of this design. A consumer can deliberately *rewind* back to an old offset and re-consume data. This violates the common contract of a queue, but proves to be an essential feature for many consumers. For example, when there is an error in application logic in the consumer, the application can re-play certain messages after the error is fixed. This is particularly important to ETL data loads into our data warehouse or Hadoop system. As another example, the consumed data may be flushed to a persistent store only periodically (e.g, a full-text indexer). If the consumer crashes, the unflushed data is lost. In this case, the consumer can checkpoint the smallest offset of the unflushed messages and re-consume from that offset when it's restarted. We note that rewinding a consumer is much easier to support in the pull model than the push model.

3.2 Distributed Coordination

We now describe how the producers and the consumers behave in a distributed setting. Each producer can publish a message to either a randomly selected partition or a partition semantically determined by a partitioning key and a partitioning function. We will focus on how the consumers interact with the brokers.

Kafka has the concept of *consumer groups*. Each consumer group consists of one or more consumers that jointly consume a set of subscribed topics, i.e., each message is delivered to only one of the consumers within the group. Different consumer groups each independently consume the full set of subscribed messages and no coordination is needed across consumer groups. The consumers

within the same group can be in different processes or on different machines. Our goal is to divide the messages stored in the brokers evenly among the consumers, without introducing too much coordination overhead.

Our first decision is to make a partition within a topic the smallest unit of parallelism. This means that at any given time, all messages from one partition are consumed only by a single consumer within each consumer group. Had we allowed multiple consumers to simultaneously consume a single partition, they would have to coordinate who consumes what messages, which necessitates locking and state maintenance overhead. In contrast, in our design consuming processes only need co-ordinate when the consumers rebalance the load, an infrequent event. In order for the load to be truly balanced, we require many more partitions in a topic than the consumers in each group. We can easily achieve this by over partitioning a topic.

The second decision that we made is to not have a central "master" node, but instead let consumers coordinate among themselves in a decentralized fashion. Adding a master can complicate the system since we have to further worry about master failures. To facilitate the coordination, we employ a highly available consensus service Zookeeper [10]. Zookeeper has a very simple, file system like API. One can create a path, set the value of a path, read the value of a path, delete a path, and list the children of a path. It does a few more interesting things: (a) one can register a watcher on a path and get notified when the children of a path or the value of a path has changed; (b) a path can be created as *ephemeral* (as oppose to *persistent*), which means that if the creating client is gone, the path is automatically removed by the Zookeeper server; (c) zookeeper replicates its data to multiple servers, which makes the data highly reliable and available.

Kafka uses Zookeeper for the following tasks: (1) detecting the addition and the removal of brokers and consumers, (2) triggering a rebalance process in each consumer when the above events happen, and (3) maintaining the consumption relationship and keeping track of the consumed offset of each partition. Specifically, when each broker or consumer starts up, it stores its information in a broker or consumer registry in Zookeeper. The broker registry contains the broker's host name and port, and the set of topics and partitions stored on it. The consumer registry includes the consumer group to which a consumer belongs and the set of topics that it subscribes to. Each consumer group is associated with an ownership registry and an offset registry in Zookeeper. The ownership registry has one path for every subscribed partition and the path value is the id of the consumer currently consuming from this partition (we use the terminology that the consumer *owns* this partition). The offset registry stores for each subscribed partition, the offset of the last consumed message in the partition.

The paths created in Zookeeper are ephemeral for the broker registry, the consumer registry and the ownership registry, and persistent for the offset registry. If a broker fails, all partitions on it are automatically removed from the broker registry. The failure of a consumer causes it to lose its entry in the consumer registry and all partitions that it owns in the ownership registry. Each consumer registers a Zookeeper watcher on both the broker registry and the consumer registry, and will be notified whenever a change in the broker set or the consumer group occurs.

During the initial startup of a consumer or when the consumer is notified about a broker/consumer change through the watcher, the consumer initiates a rebalance process to determine the new subset of partitions that it should consume from. The process is described in Algorithm 1. By reading the broker and the consumer registry from Zookeeper, the consumer first computes the set (P_T) of partitions available for each subscribed topic T and the set (C_T) of consumers subscribing to T. It then range-partitions P_T into $|C_T|$ chunks and deterministically picks one chunk to own. For each partition the consumer picks, it writes itself as the new owner of the partition in the ownership registry. Finally, the consumer begins a thread to pull data from each owned partition, starting from the offset stored in the offset registry. As messages get pulled from a partition, the consumer periodically updates the latest consumed offset in the offset registry.

Algorithm 1: rebalance process for consumer C_i in group G

```
For each topic T that C_i subscribes to {
  remove partitions owned by C_i from the ownership registry
  read the broker and the consumer registries from Zookeeper
  compute P_T = partitions available in all brokers under topic T
  compute C_T =  all consumers in G that subscribe to topic T
  sort P_T and C_T
  let j be the index position of C_i in C_T and let N = |P_T|/|C_T|
  assign partitions from j*N to (j+1)*N - 1 in P_T to consumer C_i
  for each assigned partition p {
    set the owner of p to C_i in the ownership registry
    let O_p = the offset of partition p stored in the offset registry
    invoke a thread to pull data in partition p from offset O_p
  }
}
```

When there are multiple consumers within a group, each of them will be notified of a broker or a consumer change. However, the notification may come at slightly different times at the consumers. So, it is possible that one consumer tries to take ownership of a partition still owned by another consumer. When this happens, the first consumer simply releases all the partitions that it currently owns, waits a bit and retries the rebalance process. In practice, the rebalance process often stabilizes after only a few retries.

When a new consumer group is created, no offsets are available in the offset registry. In this case, the consumers will begin with either the smallest or the largest offset (depending on a configuration) available on each subscribed partition, using an API that we provide on the brokers.

3.3 Delivery Guarantees

In general, Kafka only guarantees at-least-once delivery. Exactly-once delivery typically requires two-phase commits and is not necessary for our applications. Most of the time, a message is delivered exactly once to each consumer group. However, in the case when a consumer process crashes without a clean shutdown, the consumer process that takes over those partitions owned by the failed consumer may get some duplicate messages that are after the last offset successfully committed to zookeeper. If an application cares about duplicates, it must add its own de-duplication logic, either using the offsets that we return to the consumer or some unique key within the message. This is usually a more cost-effective approach than using two-phase commits.

Kafka guarantees that messages from a single partition are delivered to a consumer in order. However, there is no guarantee on the ordering of messages coming from different partitions.

To avoid log corruption, Kafka stores a CRC for each message in the log. If there is any I/O error on the broker, Kafka runs a recovery process to remove those messages with inconsistent CRCs. Having the CRC at the message level also allows us to check network errors after a message is produced or consumed.

If a broker goes down, any message stored on it not yet consumed becomes unavailable. If the storage system on a broker is permanently damaged, any unconsumed message is lost forever. In the future, we plan to add built-in replication in Kafka to redundantly store each message on multiple brokers.

4. Kafka Usage at LinkedIn

In this section, we describe how we use Kafka at LinkedIn. Figure 3 shows a simplified version of our deployment. We have one Kafka cluster co-located with each datacenter where our user-facing services run. The frontend services generate various kinds of log data and publish it to the local Kafka brokers in batches. We rely on a hardware load-balancer to distribute the publish requests to the set of Kafka brokers evenly. The online consumers of Kafka run in services within the same datacenter.

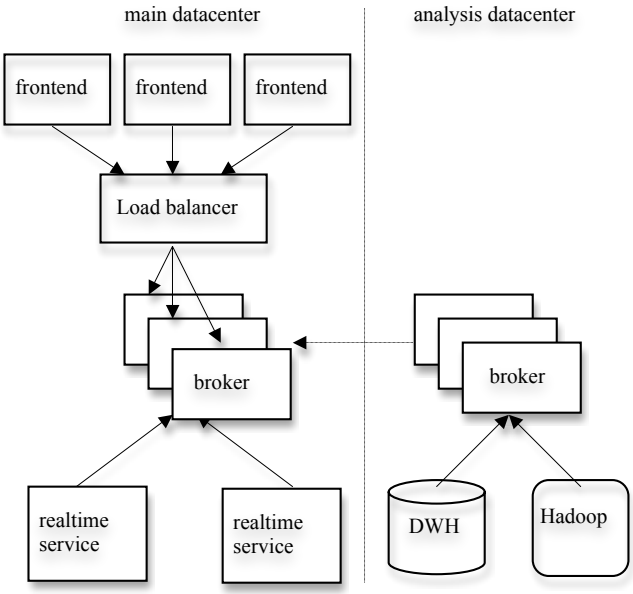


Figure 3. Kafka Deployment

We also deploy a cluster of Kafka in a separate datacenter for offline analysis, located geographically close to our Hadoop cluster and other data warehouse infrastructure. This instance of Kafka runs a set of embedded consumers to pull data from the Kafka instances in the live datacenters. We then run data load jobs to pull data from this replica cluster of Kafka into Hadoop and our data warehouse, where we run various reporting jobs and analytical process on the data. We also use this Kafka cluster for prototyping and have the ability to run simple scripts against the raw event streams for ad hoc querying. Without too much tuning, the end-to-end latency for the complete pipeline is about 10 seconds on average, good enough for our requirements.

Currently, Kafka accumulates hundreds of gigabytes of data and close to a billion messages per day, which we expect will grow significantly as we finish converting legacy systems to take advantage of Kafka. More types of messages will be added in the future. The rebalance process is able to automatically redirect the consumption when the operation staffs start or stop brokers for software or hardware maintenance.

Our tracking also includes an auditing system to verify that there is no data loss along the whole pipeline. To facilitate that, each message carries the timestamp and the server name when they are generated. We instrument each producer such that it periodically generates a monitoring event, which records the number of messages published by that producer for each topic within a fixed time window. The producer publishes the monitoring events to Kafka in a separate topic. The consumers can then count the number of messages that they have received from a given topic and validate those counts with the monitoring events to validate the correctness of data.

Loading into the Hadoop cluster is accomplished by implementing a special Kafka input format that allows MapReduce jobs to directly read data from Kafka. A MapReduce job loads the raw data and then groups and compresses it for efficient processing in the future. The stateless broker and client-side storage of message offsets again come into play here, allowing the MapReduce task management (which allows tasks to fail and be restarted) to handle the data load in a natural way without duplicating or losing messages in the event of a task restart. Both data and offsets are stored in HDFS only on the successful completion of the job.

We chose to use Avro [2] as our serialization protocol since it is efficient and supports schema evolution. For each message, we store the id of its Avro schema and the serialized bytes in the payload. This schema allows us to enforce a contract to ensure compatibility between data producers and consumers. We use a lightweight schema registry service to map the schema id to the actual schema. When a consumer gets a message, it looks up in the schema registry to retrieve the schema, which is used to decode the bytes into an object (this lookup need only be done once per schema, since the values are immutable).

5. Experimental Results

We conducted an experimental study, comparing the performance of Kafka with Apache ActiveMQ v5.4 [1], a popular open-source implementation of JMS, and RabbitMQ v2.4 [16], a message system known for its performance. We used ActiveMQ's default persistent message store KahaDB. Although not presented here, we also tested an alternative AMQ message store and found its performance very similar to that of KahaDB. Whenever possible, we tried to use comparable settings in all systems.

We ran our experiments on 2 Linux machines, each with 8 2GHz cores, 16GB of memory, 6 disks with RAID 10. The two machines are connected with a 1Gb network link. One of the machines was used as the broker and the other machine was used as the producer or the consumer.

Producer Test: We configured the broker in all systems to asynchronously flush messages to its persistence store. For each system, we ran a single producer to publish a total of 10 million messages, each of 200 bytes. We configured the Kafka producer to send messages in batches of size 1 and 50. ActiveMQ and RabbitMQ don't seem to have an easy way to batch messages and we assume that it used a batch size of 1. The results are shown in Figure 4. The x-axis represents the amount of data sent to the broker over time in MB, and the y-axis corresponds to the producer throughput in messages per second. On average, Kafka can publish messages at the rate of 50,000 and 400,000 messages per second for batch size of 1 and 50, respectively. These numbers

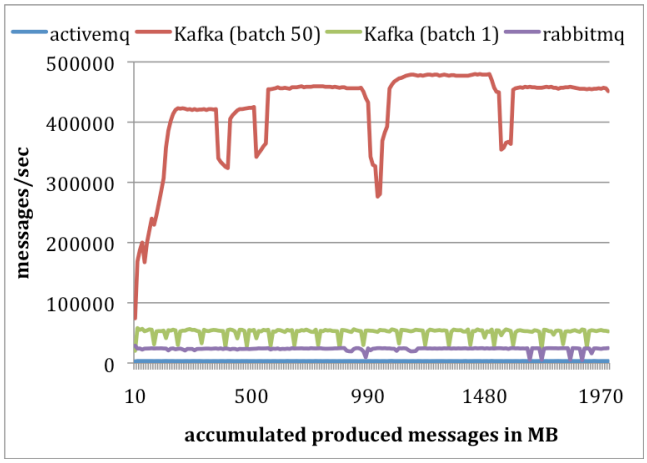


Figure 4. Producer Performance

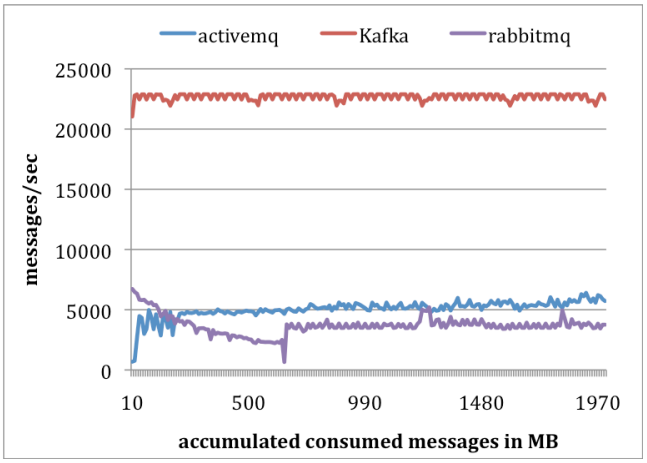


Figure 5. Consumer Performance

are orders of magnitude higher than that of ActiveMQ, and at least 2 times higher than RabbitMQ.

There are a few reasons why Kafka performed much better. First, the Kafka producer currently doesn't wait for acknowledgements from the broker and sends messages as faster as the broker can handle. This significantly increased the throughput of the publisher. With a batch size of 50, a single Kafka producer almost saturated the 1Gb link between the producer and the broker. This is a valid optimization for the log aggregation case, as data must be sent asynchronously to avoid introducing any latency into the live serving of traffic. We note that without acknowledging the producer, there is no guarantee that every published message is actually received by the broker. For many types of log data, it is desirable to trade durability for throughput, as long as the number of dropped messages is relatively small. However, we do plan to address the durability issue for more critical data in the future.

Second, Kafka has a more efficient storage format. On average, each message had an overhead of 9 bytes in Kafka, versus 144 bytes in ActiveMQ. This means that ActiveMQ was using 70% more space than Kafka to store the same set of 10 million messages. One overhead in ActiveMQ came from the heavy message header, required by JMS. Another overhead was the cost of maintaining various indexing structures. We observed that one of the busiest threads in ActiveMQ spent most of its time accessing a B-Tree to maintain message metadata and state. Finally, batching greatly improved the throughput by amortizing the RPC overhead. In Kafka, a batch size of 50 messages improved the throughput by almost an order of magnitude.

Consumer Test: In the second experiment, we tested the performance of the consumer. Again, for all systems, we used a single consumer to retrieve a total of 10 millions messages. We configured all systems so that each pull request should prefetch approximately the same amount data---up to 1000 messages or about 200KB. For both ActiveMQ and RabbitMQ, we set the consumer acknowledge mode to be automatic. Since all messages fit in memory, all systems were serving data from the page cache of the underlying file system or some in-memory buffers. The results are presented in Figure 5.

On average, Kafka consumed 22,000 messages per second, more than 4 times that of ActiveMQ and RabbitMQ. We can think of several reasons. First, since Kafka has a more efficient storage format, fewer bytes were transferred from the broker to the consumer in Kafka. Second, the broker in both ActiveMQ and RabbitMQ had to maintain the delivery state of every message. We observed that one of the ActiveMQ threads was busy writing KahaDB pages to disks during this test. In contrast, there were no disk write activities on the Kafka broker. Finally, by using the sendfile API, Kafka reduces the transmission overhead.

We close the section by noting that the purpose of the experiment is not to show that other messaging systems are inferior to Kafka. After all, both ActiveMQ and RabbitMQ have more features than Kafka. The main point is to illustrate the potential performance gain that can be achieved by a specialized system.

6. Conclusion and Future Works

We present a novel system called Kafka for processing huge volume of log data streams. Like a messaging system, Kafka employs a pull-based consumption model that allows an application to consume data at its own rate and rewind the consumption whenever needed. By focusing on log processing applications, Kafka achieves much higher throughput than conventional messaging systems. It also provides integrated distributed support and can scale out. We have been using Kafka successfully at LinkedIn for both offline and online applications.

There are a number of directions that we'd like to pursue in the future. First, we plan to add built-in replication of messages across multiple brokers to allow durability and data availability guarantees even in the case of unrecoverable machine failures. We'd like to support both asynchronous and synchronous replication models to allow some tradeoff between producer latency and the strength of the guarantees provided. An application can choose the right level of redundancy based on its requirement on durability, availability and throughput. Second, we want to add some stream processing capability in Kafka. After retrieving messages from Kafka, real time applications often perform similar operations such as window-based counting and joining each message with records in a secondary store or with messages in another stream. At the lowest level this is supported by semantically partitioning messages on the join key during publishing so that all messages sent with a particular key go to the same partition and hence arrive at a single consumer process. This provides the foundation for processing distributed streams across a cluster of consumer machines. On top of this we feel a library of helpful stream utilities, such as different windowing functions or join techniques will be beneficial to this kind of applications.

7. REFERENCES

[1] http://activemq.apache.org/

[2] http://avro.apache.org/

[3] Cloudera's Flume, https://github.com/cloudera/flume

[4] http://developer.yahoo.com/blogs/hadoop/posts/2010/06/enabling_hadoop_batch_processi_1/

[5] Efficient data transfer through zero copy: https://www.ibm.com/developerworks/linux/library/j-zerocopy/

[6] Facebook's Scribe, http://www.facebook.com/note.php?note_id=32008268919

[7] IBM Websphere MQ: http://www-01.ibm.com/software/integration/wmq/

[8] http://hadoop.apache.org/

[9] http://hadoop.apache.org/hdfs/

[10] http://hadoop.apache.org/zookeeper/

[11] http://www.slideshare.net/cloudera/hw09-hadoop-based-data-mining-platform-for-the-telecom-industry

[12] http://www.slideshare.net/prasadc/hive-percona-2009

[13] https://issues.apache.org/jira/browse/ZOOKEEPER-775

[14] JAVA Message Service: http://download.oracle.com/javaee/1.3/jms/tutorial/1_3_1-fcs/doc/jms_tutorialTOC.html.

[15] Oracle Enterprise Messaging Service: http://www.oracle.com/technetwork/middleware/ias/index-093455.html

[16] http://www.rabbitmq.com/

[17] TIBCO Enterprise Message Service: http://www.tibco.com/products/soa/messaging/

[18] Kafka, http://sna-projects.com/kafka/

Chapter 20

A simple totally ordered broadcast protocol

Authors:
Benjamin Reed, Flavio P. Junqueira

Publication year: 2008

A simple totally ordered broadcast protocol

Benjamin Reed
Yahoo! Research
Santa Clara, CA - USA
breed@yahoo-inc.com

Flavio P. Junqueira
Yahoo! Research
Barcelona, Catalunya - Spain
fpj@yahoo-inc.com

ABSTRACT

This is a short overview of a totally ordered broadcast protocol used by ZooKeeper, called Zab. It is conceptually easy to understand, is easy to implement, and gives high performance. In this paper we present the requirements ZooKeeper makes on Zab, we show how the protocol is used, and we give an overview of how the protocol works.

1. INTRODUCTION

At Yahoo! we have developed a high-performance highly-available coordination service called *ZooKeeper* [9] that allows large scale applications to perform coordination tasks such as leader election, status propagation, and rendezvous. This service implements a hierarchical space of data nodes, called *znodes*, that clients use to implement their coordination tasks. We have found the service to be flexible with performance that easily meets the production demands of the web-scale, mission critical applications we have at Yahoo!. ZooKeeper foregoes locks and instead implements wait-free shared data objects with strong guarantees on the order of operations over these objects. Client libraries take advantage of these guarantees to implement their coordination tasks. In general, one of the main premises of ZooKeeper is that order of updates is more important to applications than other typical coordination techniques such as blocking.

Embedded into ZooKeeper is a totally ordered broadcast protocol: Zab. Ordered broadcast is crucial when implementing our client guarantees; it is also necessary to maintain replicas of the ZooKeeper state at each ZooKeeper server. These replicas stay consistent using our totally ordered broadcast protocol, such as with replicated state-machines [13]. This paper focuses on the requirements ZooKeeper makes on this broadcast protocol and an overview of its implementation.

A ZooKeeper service usually consists of three to seven machines. Our implementation supports more machines, but three to seven machines provide more than enough performance and resilience. A client connects to any of the machines providing the service and always has a consistent view of the ZooKeeper state. The service tolerates up to f crash failures, and it requires at least $2f + 1$ servers.

Applications use ZooKeeper extensively and have tens to thousands of clients accessing it concurrently, so we require high throughput. We have designed ZooKeeper for workloads with ratios of read to write operations that are higher than 2:1; however, we have found that ZooKeeper's high write throughput allows it to be used for some write dominant workloads as well. ZooKeeper provides high read throughput by servicing the reads from the local replica of the ZooKeeper state at each server. As a consequence, both fault tolerance and read throughput scales by adding servers to the service. Write throughput does not scale by adding servers; instead it is limited by the throughput of the broadcast protocol, thus we need a broadcast protocol with high-throughput.

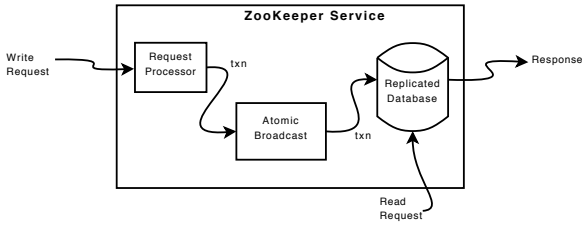


Figure 1: The logical components of the ZooKeeper service.

Figure 1 shows the logical makeup of the ZooKeeper service. Read requests are serviced from the local database containing the ZooKeeper state. Write requests are transformed from ZooKeeper requests to idempotent transactions and sent through Zab before a response is generated. Many ZooKeeper write requests are conditional in nature: a znode can only be deleted if it does not have any children; a znode can be created with a name and a sequence number appended to it; a change to data will only be applied if it is at an expected version. Even the non-conditional write requests modify meta data, such as version numbers, in a ways that are not idempotent.

By sending all updates through a single server, referred to as the leader, we transform non-idempotent requests into idempotent transactions. We use the term *transaction* to denote the idempotent version of a request throughout this paper. The leader can perform the transformation because it has a perfect view of the future state of the replicated

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...$5.00.

database and can calculate the state of the new record. The idempotent transaction is a record of this new state. ZooKeeper takes advantage of the idempotent transactions in many ways that are out of the scope of this paper, but the idempotent nature of our transactions also allows us to relax an ordering requirement of our broadcast protocol during recovery.

2. REQUIREMENTS

We assume a set of processes that both implement the atomic broadcast protocol and use it. To guarantee the correct transformation into idempotent requests in ZooKeeper, it is necessary that there is a single leader at a time, so we we enforce that there is one such a process through the protocol implementation. We discuss it more when we present more detail of the protocol.

ZooKeeper makes the following requirements on the broadcast protocol:

Reliable delivery If a message, m, is delivered by one server, then it will be eventually delivered by all correct servers.

Total order If message a is delivered before message b by one server, then every server that delivers a and b delivers a before b.

Causal order If message a causally precedes message b and both messages are delivered, then a must be ordered before b.

For correctness, ZooKeeper additionally requires the following prefix property:

Prefix property: If m is the last message delivered for a leader L, any message proposed before m by L must also be delivered;

Note that a process might be elected multiple times. Each time, however, counts as a different leader for the purposes of this prefix property.

With these three guarantees we can maintain correct replicas of the ZooKeeper database:

1. The reliability and total order guarantees ensure that all of the replicas have a consistent state;
2. The causal order ensures that the replicas have state correct from the perspective of the application using Zab;
3. The leader proposes updates to the database based on requests received.

It is important to observe that there are two types of causal relationships taken into account by Zab:

1. If two messages, a and b, are sent by the same server and a is proposed before b, we say that a causally precedes b;
2. Zab assumes a single leader server at a time that can commit proposals. If a leader changes, any previously proposed messages causally precede messages proposed by the new leader.

Causal violation example.
To show the problems caused by violating the second dimension of our causal relationship consider the following scenario:

- A ZooKeeper client C_1 requests to set a znode `/a` to 1 which will be transformed into a message, w_1, that contains ("/a", "1", 1), where the tuple represents path, value, and resulting version of the znode.
- C_1 then requests to set `/a` to 2 which will be transformed into a message, w_2, that contains ("/a", "2", 2);
- L_1 proposes and delivers w_1, but it is only able to issue w_2 to itself before failing;
- A new leader L_2 takes over.
- A client C_2 requests to set `/a` to 3 conditioned on a being at version 1, which will be transformed into a message, w_3, that contains ("/a", "3", 2);
- L_2 proposes and delivers w_3.

In this scenario, the client receives a successful response to w_1, but an error for w_2 because of the leader failure. If eventually L_1 recovers, regains leadership, and attempts to deliver the proposal for w_2, the causal ordering of the client requests will be violated and the replicated state will be incorrect.

Our failure model is crash-fail with stateful recovery. We do not assume synchronized clocks, but we do assume that servers perceive time pass at approximately the same rate. (We use timeouts to detect failures.) The processes that make up Zab have persistent state, so processes can restart after a failure and recover using the persistent state. This means a process can have state that is partially valid such as missing more recent transactions or, more problematic, the process may have transactions that were never delivered earlier and now must be skipped.

We provision to handle f failures, but we also need to handle correlated recoverable failures, for example power outages. To recover from these failures we require that messages are recorded on the disk media of a quorum of disks before a message is delivered. (Very non-technical, pragmatic, operations-oriented reasons prevent us from incorporating devices like UPS devices, redundant/dedicated networking devices, and NVRAMs into our design.)

Although we do not assume Byzantine failures, we do detect data corruption using digests. We also add extra metadata to protocol packets and use them for sanity checks. We abort the server process if we detect data corruption or sanity checks fail.

Practical issues of independent implementations of operating environments and of the protocols themselves make the realization of fully Byzantine tolerant systems impractical for our application. It has also been shown that achieving truly reliable independent implementations requires more than just programming resources [11]. To date, most of our production problems have been either the result of implementation bugs that affect all replicas or problems that

are outside the implementation scope of Zab, but affect Zab, such as network misconfiguration.

ZooKeeper uses an in-memory database and stores transaction logs and periodic snapshots on disk. Zab's transaction log doubles as the database write-ahead transaction log so that a transaction will only be written to the disk once. Since the database is in-memory and we use gigabit interface cards, the bottleneck is the disk I/O on the writes. To mitigate the disk I/O bottleneck we write transactions in batches so that we can record multiple transactions in a single write to the disk. This batching happens at the replicas not at the the protocol level, so the implementation is much closer to group commits [4, 6] from databases than message packing [5]. We chose to not use message packing to minimize latency, while still getting most of the benefits of packing through batched I/O to the disk.

Our crash-fail with stateful recovery model means that when a server recovers, it is going to read in its snapshot and replay all delivered transactions after that snapshot. Thus, during recovery the atomic broadcast does not need to guarantee at most once delivery. Our use of idempotent transactions means that multiple delivery of a transaction is fine as long as on restart order is preserved. This is a relaxation of the total order requirement. Specifically, if a is delivered before b and after a failure a is redelivered, b will also be redelivered after a.

Other performance requirements we have are:

Low latency: ZooKeeper is used extensively by applications and our users expect low response times.

Bursty high throughput: Applications using ZooKeeper usually have read-dominant workloads, but occasionally radical reconfigurations occur that cause large spikes in write throughput.

Smooth failure handling: If a server fails that is not the leader and there is still a quorum of correct servers, there should be no service interruption.

3. WHY ANOTHER PROTOCOL

Reliable broadcast protocols can present different semantics depending on the application requirements. For example, Birman and Joseph propose two primitives, ABCAST and CBCAST, that satisfy total order and causal order, respectively [2]. Zab also provides causal and total ordering guarantees.

A good protocol candidate in our case is *Paxos* [12]. Paxos has a number of important properties, such as guaranteeing safety despite the number of process failures, allowing processes to crash and recover, and enabling operations to commit within three communication steps under realistic assumptions. We observe that there are a couple of realistic assumptions we can make that enable us to simplify Paxos and obtain high throughput. First, Paxos tolerates message losses and reordering of messages. By using TCP to communicate between pairs of servers, we can guarantee that delivered messages are delivered in FIFO order, which enables us to satisfy per proposer causality even when server processes have multiple outstanding proposed messages. Paxos, however, does not directly guarantee causality because it does not require FIFO channels.

Proposers are the agents proposing values for the different instances in Paxos. To guarantee progress, there must be a single proposer proposing, otherwise proposers may contend forever on a given instance. Such an active proposer is a leader. When Paxos recovers from a leader failure, the new leader makes sure all partially delivered messages are fully delivered and then resumes proposals from the instance number the old leader left off. Because multiple leaders can propose a value for a given instance two problems arise. First, proposals can conflict. Paxos uses ballots to detect and resolve conflicting proposals. Second, it is not enough to know that a given instance number has been committed, processes must also be able to figure out which value has been committed. Zab avoids both of these problems by ensuring that there is only one message proposal for a given proposal number. This obviates the need for ballots and simplifies recovery. In Paxos, if a server believes it is a leader, it will use a higher ballot to take over leadership from a previous leader. However, in Zab a new leader cannot take over leadership until a quorum of servers abandon the previous leader.

An alternative way of obtaining high throughput is by limiting the protocol message complexity per broadcast message such as with the Fixed-Sequencer Ring (FSR) protocol [8]. Throughput does not decrease with FSR as the system grows, but latency does increase with the number of processes, which is inconvenient in our environment. Virtual synchrony also enables high throughput when groups are stable for long enough [1]. However, any server failure triggers a reconfiguration of the service, thus causing brief service interruptions during such reconfigurations. Moreover, a failure detector in such systems has to monitor all servers. The reliability of such a failure detector is crucial for the stability and speed of reconfiguration. Leader-based protocols also rely upon failure detectors for liveness, but such a failure detector only monitors one server at a time, which is the leader. As we discuss in the next section, we do not use fixed quorums or groups for writes and maintain the service availability as long as the leader does not fail.

Our protocol has a fixed sequencer, according to the classification of Defago *et al.* [3], that we call the leader. Such a leader is elected through a leader election algorithm and synchronized with a quorum of other servers, called followers. Since the leader has to manage messages to all the followers this decision of a fixed sequencer distributes load unevenly across the servers that compose the system with respect to the broadcast protocol. We have taken this approach, though, for the following reasons:

- Clients can connect to any server, and servers have to serve read operations locally and maintain information about the session of a client. This extra load of a follower process (a process that is not a leader) makes the load more evenly distributed;
- The number of servers involved is small. This means that the network communication overhead does not become the bottleneck that can affect fixed sequencer protocols;
- Implementing more complex approaches was not necessary as this simple one provides enough performance.

Having a moving sequencer, for example, increases the complexity of the implementation as we have to handle problems such as losing the token. Also, we opted to move away from models based on communication history, such as sender-based ones, to avoid the quadratic message complexity of such protocols. Destination agreement protocols suffer from a similar problem [8].

Using a leader requires that we recover from leader failures

to guarantee progress. We use techniques related to view changes such as in the protocol of Keidar and Dolev [10]. Different from their protocol, we do not operate using group communication. If a new server joins or leaves (perhaps by crashing), then we do no cause a view change, unless such an event corresponds to the leader crashing.

4. PROTOCOL

Zab's protocol consists of two modes: recovery and broadcast. When the service starts or after a leader failure, Zab transitions to recovery mode. Recovery mode ends when a leader emerges and a quorum of servers have synchronized their state with the leader. Synchronizing their state consists of guaranteeing that the leader and new server have the same state.

Once a leader has a quorum of synchronized followers, it can begin to broadcast messages. As we mentioned in the introduction, the ZooKeeper service itself uses a leader to process requests. The leader is the server that executes a broadcast by initiating the broadcast protocol, and any server other than the leader that needs to broadcast a message first forwards it to the leader. By using the leader that emerges from the recovery mode as both the leader to process write requests and to coordinate the broadcast protocol, we eliminate the network latency of forwarding messages to broadcast from the write request leader to the broadcast protocol leader.

Once a leader has synchronized with a quorum of followers, it begins to broadcast messages. If a Zab server comes online while a leader is actively broadcasting messages, the server will start in recovery mode, discover and synchronize with the leader, and start participating in the message broadcasts. The service remains in broadcast mode until the leader fails or it no longer has a quorum of followers. Any quorum of followers are sufficient for a leader and the service to stay active. For example, a Zab service made up of three servers where one is a leader and the two other servers are followers will move to broadcast mode. If one of the followers die, there will be no interruption in service since the leader will still have a quorum. If the follower recovers and the other dies, there will still be no service interruption.

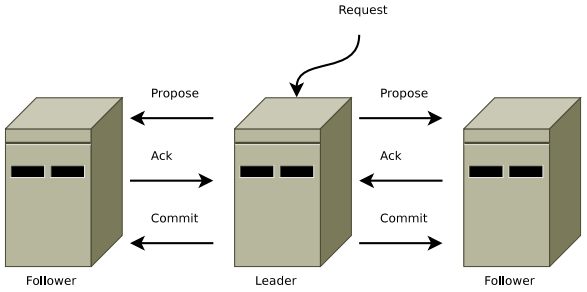


Figure 2: The protocol for message delivery. Multiple instances of this protocol can be running concurrently. The message is delivered once the leader issues the COMMIT.

4.1 Broadcast

The protocol we use while the atomic broadcast service is operational, called broadcast mode, resembles a simple a two-phase commit [7]: a leader proposes a request, collects votes, and finally commits. Figure 2 illustrates the flow of messages with our protocol. We are able to simplify the two-phase commit protocol because we do not have aborts; followers either acknowledge the leader's proposal or they abandon the leader. The lack of aborts also mean that we can commit once a quorum of servers ack the proposal rather than waiting for all servers to respond. This simplified two-phase commit by itself cannot handle leader failures, so we will add recovery mode to handle leader failures.

The broadcast protocol uses FIFO (TCP) channels for all communications. By using FIFO channels, preserving the ordering guarantees becomes very easy. Messages are delivered in order through FIFO channels; as long as messages are processed as they are received, order is preserved.

The leader broadcasts a proposal for a message to be delivered. Before proposing a message the leader assigns a monotonically increasing unique id, called the zxid. Because Zab preserves causal ordering, the delivered messages will also be ordered by their zxids. Broadcasting consists of putting the proposal with the message attached into the outgoing queue for each follower to be sent through the FIFO channel to the follower. When a follower receives a proposal, it writes it to disk, batching when it can, and sends an acknowledgement to the leader as soon as the proposal is on the disk media. When a leader receives ACKs from a quorum, the leader will broadcast a COMMIT and deliver the message locally. Followers deliver the message when they receive the COMMIT from the leader.

Note that the leader does not have to send the COMMIT if the followers broadcast the ACKs to each other. Not only does this modification increases the amount of network traffic, but it also requires a complete communication graph rather than the simple star topology, which is easier to manage from the point of view of starting TCP connections. Maintaining this graph and tracking the ACKs at the clients was also deemed an unacceptable complication in our implementation.

4.2 Recovery

This simple broadcast protocol works great until the leader fails or loses a quorum of followers. To guarantee progress, a recovery procedure is necessary to elect a new leader and bring all servers to a correct state. For leader election we need an algorithm that succeeds with high probability to guarantee liveness. The leader election protocol enables not only the leader to learn that it is the leader, but also a quorum to agree upon this decision. If the election phase completes erroneously, servers will not make progress, they eventually timeout, and restart the leader election. In our implementation we have two different implementations of leader election. The fastest completes leader election in just a few hundreds of milliseconds if there is a quorum of correct servers.

Part of the complication of the recovery procedure is the sheer number of proposals that can be in flight at a given time. The maximum number of in-flight proposals is a configurable option, but the default is one thousand. To enable such a protocol to work despite failures of the leader there are two specific guarantees we need to make: we must never forget delivered messages and we need to let go of messages that are skipped.

A message that gets delivered on one machine must be

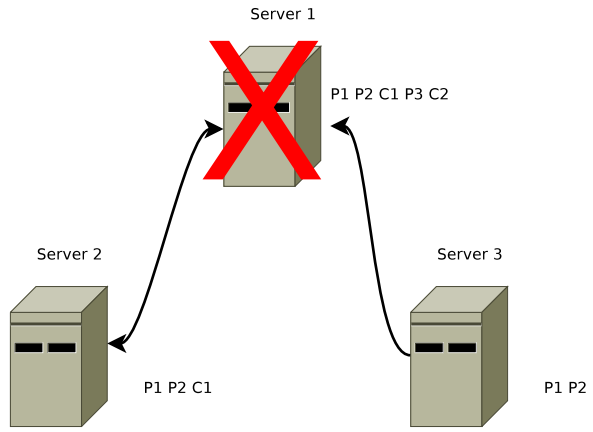


Figure 3: An example of a message that cannot be forgotten. Server 1 is the leader. It fails, and it was the only only server to see the COMMIT of message 2. The broadcast protocol must ensure that message 2 gets committed on all correct servers.

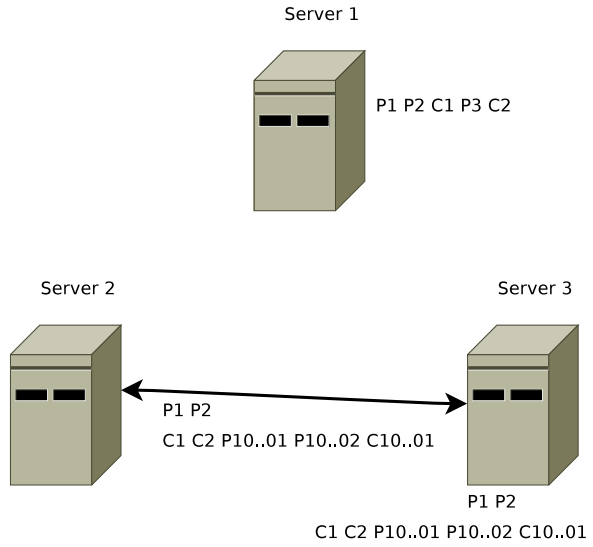


Figure 4: An example of a message that must be skipped. Server 1 has a proposal for message 3 that was not committed. Since later proposals have been committed message 3 must be forgotten.

delivered on all even if that machine fails. This situation can easily occur if the leader commits a message and then fails before the COMMIT reaches any other server as shown in Figure 3. Because the leader committed the message, a client could have seen the result of the transaction in the message, so the transaction must be delivered to all other servers eventually so that the client sees a consistent view of the service.

Conversely, a skipped message must remain skipped. Again, this situation can easily occur if a proposal gets generated by a leader and the leader fails before anyone else sees the proposal. For example, in Figure 3 no other server saw proposal number 3, so in Figure 4 when server 1 comes back up it needs to be sure to throw away proposal number 3 when it reintegrates with the system. If server 1 became a new leader and committed 3 after messages 100000001 and 100000002 have been delivered, we would violate our ordering guarantees.

The problem of remembering delivered messages is handled with a simple tweak of the leader election protocol. If the leader election protocol guarantees that the new leader has the highest proposal number in a quorum of servers, a newly elected leader will also have all committed messages. Before proposing any new messages a newly elected leader first makes sure that all messages that are in its transaction log have been proposed to and committed by a quorum of followers. Note that having the new leader being a server process with highest zxid is an optimization as a newly elected leader in this case does not need to find out from a group of followers which one contains the highest zxid and fetch missing transactions.

All servers that are operating correctly will either be a leader or be following a leader. The leader ensures that its followers have seen all the proposals and have delivered all that have been committed. It accomplishes this task by queuing to a newly connected follower any PROPOSAL it has that the follower has not seen, and then queuing a COMMIT for all such proposals up to the last message committed. After all such messages have been queued, the leader adds the follower to the broadcast list for future PROPOSALs and ACKs.

Skipping messages that have been proposed but never delivered is also simple to handle. In our implementation the zxid is a 64-bit number with the lower 32-bits used as a simple counter. Each proposal increments the counter. The high order 32-bit is the epoch. Each time a new leader takes over it will get the epoch of the highest zxid it has in its log, increment the epoch, and using a zxid with the epoch bits set to this new epoch and the counter set to zero. Using epochs to mark leadership changes and requiring that a quorum of servers recognize a server as the leader for that epoch, we avoid the possibility of multiple leaders issuing different proposals with the same zxid. One advantage of this scheme is that we can skip instances upon a leader failure, thus speeding up and simplifying the recovery process. If a server that has been down is restarted with a proposal that was never delivered from a previous epoch, it is not able to be a new leader since every possible quorum of servers has a server with a proposal from a newer epoch and therefore a higher zxid. When this server connects as a follower, the leader checks the last committed proposal for the epoch of the follower's latest proposal and tell the follower to truncate its transaction log (*i.e.*, forget) to the last committed

proposal for that epoch. In Figure 4 when server 1 connects to the leader, the leader tells it to purge proposal 3 from its transaction log.

5. FINAL REMARKS

We were able to implement this protocol quickly and it has proven to be robust in production. Most important, we met our goals of high throughput and low latency. On a non-saturated system, latencies are measured in milliseconds since the typical latencies will correspond to 4 times the packet transmission latency irrespective of the number of servers. Bursty loads are also handled gracefully since messages are committed when a quorum ack proposals. A slow server will not affect bursty throughput since fast quorums of servers that do not contain the slow server can ack the messages. Finally, since the leader will commit messages as soon as any quorum acks a proposal followers that fail do not affect the availability or even the throughput of the service as long as there is a quorum of correct servers.

As result of using efficient implementations of these protocols, we have an implementation of the system that reaches tens to hundreds of thousands of operations per second for workload mixes of 2:1 read-to-write ratios and higher. This system is currently in production use, and it is used by large applications such as the Yahoo! crawler and the Yahoo! advertisement system.

Acknowledgements

We would to thank both of our reviewers for useful comments, and Robbert van Renesse for a very nice offline discussion on Zab which helped us to clarify several points.

6. REFERENCES

[1] K. Birman and T. Joseph. Exploiting virtual synchrony in distributed systems. *SIGOPS Oper. Syst. Rev.*, 21(5):123–138, 1987.

[2] K. P. Birman and T. A. Joseph. Reliable communication in the presence of failures. *ACM Trans. Comput. Syst.*, 5(1):47–76, 1987.

[3] X. Défago, A. Schiper, and P. Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, 36(4):372–421, 2004.

[4] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. Wood. Implementation techniques for main memory database systems. *SIGMOD Rec.*, 14(2):1–8, 1984.

[5] R. Friedman and R. van Renesse. Packing messages as a tool for boosting the performance of total ordering protocols. In *HPDC*, pages 233–242, 1997.

[6] D. Gawlick and D. Kinkade. Varieties of concurrency control in ims/vs fast path. *IEEE Database Eng. Bull.*, 8(2):3–10, 1985.

[7] J. Gray. Notes on data base operating systems. In *Operating Systems, An Advanced Course*, pages 393–481, London, UK, 1978. Springer-Verlag.

[8] R. Guerraoui, R. R. Levy, B. Pochon, and V. Quema. High throughput total order broadcast for cluster environments. In *DSN '06: Proceedings of the International Conference on Dependable Systems and Networks*, pages 549–557, Washington, DC, USA, 2006. IEEE Computer Society.

[9] http://hadoop.apache.org/zookeeper. Zookeeper project page, 2008.

[10] I. Keidar and D. Dolev. Totally ordered broadcast in the face of network partitions. In D. R. Avresky, editor, *Dependable Network Computing*, chapter 3, pages 51–75. Kluwer Academic, 2000.

[11] J. C. Knight and N. G. Leveson. An experimental evaluation of the assumption of independence in multiversion programming. *IEEE Trans. Softw. Eng.*, 12(1):96–109, 1986.

[12] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.

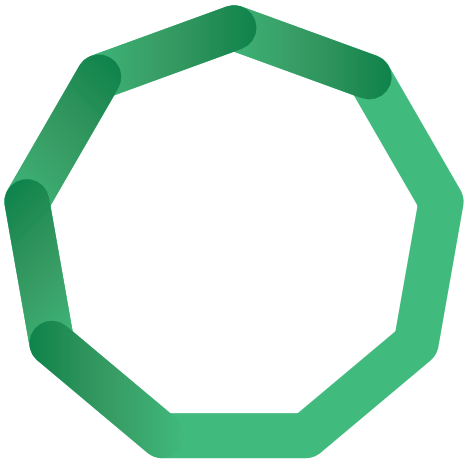
[13] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Comput. Surv.*, 22(4):299–319, 1990.

Chapter 21

In Search of an Understandable Consensus Algorithm

Authors:
Diego Ongaro, John Ousterhout

Publication year: 2014



In Search of an Understandable Consensus Algorithm (Extended Version)

Diego Ongaro and John Ousterhout
Stanford University

Abstract

Raft is a consensus algorithm for managing a replicated log. It produces a result equivalent to (multi-)Paxos, and it is as efficient as Paxos, but its structure is different from Paxos; this makes Raft more understandable than Paxos and also provides a better foundation for building practical systems. In order to enhance understandability, Raft separates the key elements of consensus, such as leader election, log replication, and safety, and it enforces a stronger degree of coherency to reduce the number of states that must be considered. Results from a user study demonstrate that Raft is easier for students to learn than Paxos. Raft also includes a new mechanism for changing the cluster membership, which uses overlapping majorities to guarantee safety.

1 Introduction

Consensus algorithms allow a collection of machines to work as a coherent group that can survive the failures of some of its members. Because of this, they play a key role in building reliable large-scale software systems. Paxos [15, 16] has dominated the discussion of consensus algorithms over the last decade: most implementations of consensus are based on Paxos or influenced by it, and Paxos has become the primary vehicle used to teach students about consensus.

Unfortunately, Paxos is quite difficult to understand, in spite of numerous attempts to make it more approachable. Furthermore, its architecture requires complex changes to support practical systems. As a result, both system builders and students struggle with Paxos.

After struggling with Paxos ourselves, we set out to find a new consensus algorithm that could provide a better foundation for system building and education. Our approach was unusual in that our primary goal was *understandability*: could we define a consensus algorithm for practical systems and describe it in a way that is significantly easier to learn than Paxos? Furthermore, we wanted the algorithm to facilitate the development of intuitions that are essential for system builders. It was important not just for the algorithm to work, but for it to be obvious why it works.

The result of this work is a consensus algorithm called Raft. In designing Raft we applied specific techniques to improve understandability, including decomposition (Raft separates leader election, log replication, and safety) and state space reduction (relative to Paxos, Raft reduces the degree of nondeterminism and the ways servers can be inconsistent with each other). A user study with 43 students at two universities shows that Raft is significantly easier to understand than Paxos: after learning both algorithms, 33 of these students were able to answer questions about Raft better than questions about Paxos.

Raft is similar in many ways to existing consensus algorithms (most notably, Oki and Liskov's Viewstamped Replication [29, 22]), but it has several novel features:

- **Strong leader:** Raft uses a stronger form of leadership than other consensus algorithms. For example, log entries only flow from the leader to other servers. This simplifies the management of the replicated log and makes Raft easier to understand.
- **Leader election:** Raft uses randomized timers to elect leaders. This adds only a small amount of mechanism to the heartbeats already required for any consensus algorithm, while resolving conflicts simply and rapidly.
- **Membership changes:** Raft's mechanism for changing the set of servers in the cluster uses a new *joint consensus* approach where the majorities of two different configurations overlap during transitions. This allows the cluster to continue operating normally during configuration changes.

We believe that Raft is superior to Paxos and other consensus algorithms, both for educational purposes and as a foundation for implementation. It is simpler and more understandable than other algorithms; it is described completely enough to meet the needs of a practical system; it has several open-source implementations and is used by several companies; its safety properties have been formally specified and proven; and its efficiency is comparable to other algorithms.

The remainder of the paper introduces the replicated state machine problem (Section 2), discusses the strengths and weaknesses of Paxos (Section 3), describes our general approach to understandability (Section 4), presents the Raft consensus algorithm (Sections 5–8), evaluates Raft (Section 9), and discusses related work (Section 10).

2 Replicated state machines

Consensus algorithms typically arise in the context of *replicated state machines* [37]. In this approach, state machines on a collection of servers compute identical copies of the same state and can continue operating even if some of the servers are down. Replicated state machines are

This tech report is an extended version of [32]; additional material is noted with a gray bar in the margin. Published May 20, 2014.

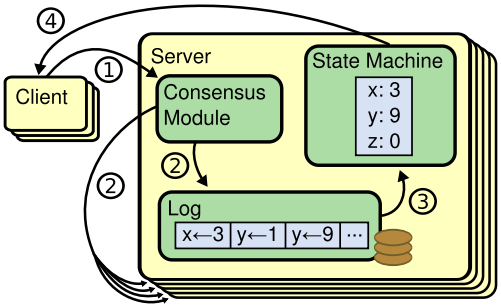


Figure 1: Replicated state machine architecture. The consensus algorithm manages a replicated log containing state machine commands from clients. The state machines process identical sequences of commands from the logs, so they produce the same outputs.

used to solve a variety of fault tolerance problems in distributed systems. For example, large-scale systems that have a single cluster leader, such as GFS [8], HDFS [38], and RAMCloud [33], typically use a separate replicated state machine to manage leader election and store configuration information that must survive leader crashes. Examples of replicated state machines include Chubby [2] and ZooKeeper [11].

Replicated state machines are typically implemented using a replicated log, as shown in Figure 1. Each server stores a log containing a series of commands, which its state machine executes in order. Each log contains the same commands in the same order, so each state machine processes the same sequence of commands. Since the state machines are deterministic, each computes the same state and the same sequence of outputs.

Keeping the replicated log consistent is the job of the consensus algorithm. The consensus module on a server receives commands from clients and adds them to its log. It communicates with the consensus modules on other servers to ensure that every log eventually contains the same requests in the same order, even if some servers fail. Once commands are properly replicated, each server's state machine processes them in log order, and the outputs are returned to clients. As a result, the servers appear to form a single, highly reliable state machine.

Consensus algorithms for practical systems typically have the following properties:

- They ensure *safety* (never returning an incorrect result) under all non-Byzantine conditions, including network delays, partitions, and packet loss, duplication, and reordering.
- They are fully functional (*available*) as long as any majority of the servers are operational and can communicate with each other and with clients. Thus, a typical cluster of five servers can tolerate the failure of any two servers. Servers are assumed to fail by stopping; they may later recover from state on stable storage and rejoin the cluster.
- They do not depend on timing to ensure the consistency of the logs: faulty clocks and extreme message delays can, at worst, cause availability problems.
- In the common case, a command can complete as soon as a majority of the cluster has responded to a single round of remote procedure calls; a minority of slow servers need not impact overall system performance.

3 What's wrong with Paxos?

Over the last ten years, Leslie Lamport's Paxos protocol [15] has become almost synonymous with consensus: it is the protocol most commonly taught in courses, and most implementations of consensus use it as a starting point. Paxos first defines a protocol capable of reaching agreement on a single decision, such as a single replicated log entry. We refer to this subset as *single-decree Paxos*. Paxos then combines multiple instances of this protocol to facilitate a series of decisions such as a log (*multi-Paxos*). Paxos ensures both safety and liveness, and it supports changes in cluster membership. Its correctness has been proven, and it is efficient in the normal case.

Unfortunately, Paxos has two significant drawbacks. The first drawback is that Paxos is exceptionally difficult to understand. The full explanation [15] is notoriously opaque; few people succeed in understanding it, and only with great effort. As a result, there have been several attempts to explain Paxos in simpler terms [16, 20, 21]. These explanations focus on the single-decree subset, yet they are still challenging. In an informal survey of attendees at NSDI 2012, we found few people who were comfortable with Paxos, even among seasoned researchers. We struggled with Paxos ourselves; we were not able to understand the complete protocol until after reading several simplified explanations and designing our own alternative protocol, a process that took almost a year.

We hypothesize that Paxos' opaqueness derives from its choice of the single-decree subset as its foundation. Single-decree Paxos is dense and subtle: it is divided into two stages that do not have simple intuitive explanations and cannot be understood independently. Because of this, it is difficult to develop intuitions about why the single-decree protocol works. The composition rules for multi-Paxos add significant additional complexity and subtlety. We believe that the overall problem of reaching consensus on multiple decisions (i.e., a log instead of a single entry) can be decomposed in other ways that are more direct and obvious.

The second problem with Paxos is that it does not provide a good foundation for building practical implementations. One reason is that there is no widely agreed-upon algorithm for multi-Paxos. Lamport's descriptions are mostly about single-decree Paxos; he sketched possible approaches to multi-Paxos, but many details are missing. There have been several attempts to flesh out and optimize Paxos, such as [26], [39], and [13], but these differ

from each other and from Lamport's sketches. Systems such as Chubby [4] have implemented Paxos-like algorithms, but in most cases their details have not been published.

Furthermore, the Paxos architecture is a poor one for building practical systems; this is another consequence of the single-decree decomposition. For example, there is little benefit to choosing a collection of log entries independently and then melding them into a sequential log; this just adds complexity. It is simpler and more efficient to design a system around a log, where new entries are appended sequentially in a constrained order. Another problem is that Paxos uses a symmetric peer-to-peer approach at its core (though it eventually suggests a weak form of leadership as a performance optimization). This makes sense in a simplified world where only one decision will be made, but few practical systems use this approach. If a series of decisions must be made, it is simpler and faster to first elect a leader, then have the leader coordinate the decisions.

As a result, practical systems bear little resemblance to Paxos. Each implementation begins with Paxos, discovers the difficulties in implementing it, and then develops a significantly different architecture. This is time-consuming and error-prone, and the difficulties of understanding Paxos exacerbate the problem. Paxos' formulation may be a good one for proving theorems about its correctness, but real implementations are so different from Paxos that the proofs have little value. The following comment from the Chubby implementers is typical:

> There are significant gaps between the description of the Paxos algorithm and the needs of a real-world system.... the final system will be based on an unproven protocol [4].

Because of these problems, we concluded that Paxos does not provide a good foundation either for system building or for education. Given the importance of consensus in large-scale software systems, we decided to see if we could design an alternative consensus algorithm with better properties than Paxos. Raft is the result of that experiment.

4 Designing for understandability

We had several goals in designing Raft: it must provide a complete and practical foundation for system building, so that it significantly reduces the amount of design work required of developers; it must be safe under all conditions and available under typical operating conditions; and it must be efficient for common operations. But our most important goal—and most difficult challenge—was *understandability*. It must be possible for a large audience to understand the algorithm comfortably. In addition, it must be possible to develop intuitions about the algorithm, so that system builders can make the extensions that are inevitable in real-world implementations.

There were numerous points in the design of Raft where we had to choose among alternative approaches. In these situations we evaluated the alternatives based on understandability: how hard is it to explain each alternative (for example, how complex is its state space, and does it have subtle implications?), and how easy will it be for a reader to completely understand the approach and its implications?

We recognize that there is a high degree of subjectivity in such analysis; nonetheless, we used two techniques that are generally applicable. The first technique is the well-known approach of problem decomposition: wherever possible, we divided problems into separate pieces that could be solved, explained, and understood relatively independently. For example, in Raft we separated leader election, log replication, safety, and membership changes.

Our second approach was to simplify the state space by reducing the number of states to consider, making the system more coherent and eliminating nondeterminism where possible. Specifically, logs are not allowed to have holes, and Raft limits the ways in which logs can become inconsistent with each other. Although in most cases we tried to eliminate nondeterminism, there are some situations where nondeterminism actually improves understandability. In particular, randomized approaches introduce nondeterminism, but they tend to reduce the state space by handling all possible choices in a similar fashion ("choose any; it doesn't matter"). We used randomization to simplify the Raft leader election algorithm.

5 The Raft consensus algorithm

Raft is an algorithm for managing a replicated log of the form described in Section 2. Figure 2 summarizes the algorithm in condensed form for reference, and Figure 3 lists key properties of the algorithm; the elements of these figures are discussed piecewise over the rest of this section.

Raft implements consensus by first electing a distinguished *leader*, then giving the leader complete responsibility for managing the replicated log. The leader accepts log entries from clients, replicates them on other servers, and tells servers when it is safe to apply log entries to their state machines. Having a leader simplifies the management of the replicated log. For example, the leader can decide where to place new entries in the log without consulting other servers, and data flows in a simple fashion from the leader to other servers. A leader can fail or become disconnected from the other servers, in which case a new leader is elected.

Given the leader approach, Raft decomposes the consensus problem into three relatively independent subproblems, which are discussed in the subsections that follow:

- **Leader election:** a new leader must be chosen when an existing leader fails (Section 5.2).
- **Log replication:** the leader must accept log entries

State

Persistent state on all servers:
(Updated on stable storage before responding to RPCs)

currentTerm	latest term server has seen (initialized to 0 on first boot, increases monotonically)
votedFor	candidateId that received vote in current term (or null if none)
log[]	log entries; each entry contains command for state machine, and term when entry was received by leader (first index is 1)

Volatile state on all servers:

commitIndex	index of highest log entry known to be committed (initialized to 0, increases monotonically)
lastApplied	index of highest log entry applied to state machine (initialized to 0, increases monotonically)

Volatile state on leaders:
(Reinitialized after election)

nextIndex[]	for each server, index of the next log entry to send to that server (initialized to leader last log index + 1)
matchIndex[]	for each server, index of highest log entry known to be replicated on server (initialized to 0, increases monotonically)

AppendEntries RPC

Invoked by leader to replicate log entries (§5.3); also used as heartbeat (§5.2).

Arguments:

term	leader's term
leaderId	so follower can redirect clients
prevLogIndex	index of log entry immediately preceding new ones
prevLogTerm	term of prevLogIndex entry
entries[]	log entries to store (empty for heartbeat; may send more than one for efficiency)
leaderCommit	leader's commitIndex

Results:

term	currentTerm, for leader to update itself
success	true if follower contained entry matching prevLogIndex and prevLogTerm

Receiver implementation:

1. Reply false if term < currentTerm (§5.1)
2. Reply false if log doesn't contain an entry at prevLogIndex whose term matches prevLogTerm (§5.3)
3. If an existing entry conflicts with a new one (same index but different terms), delete the existing entry and all that follow it (§5.3)
4. Append any new entries not already in the log
5. If leaderCommit > commitIndex, set commitIndex = min(leaderCommit, index of last new entry)

RequestVote RPC

Invoked by candidates to gather votes (§5.2).

Arguments:

term	candidate's term
candidateId	candidate requesting vote
lastLogIndex	index of candidate's last log entry (§5.4)
lastLogTerm	term of candidate's last log entry (§5.4)

Results:

term	currentTerm, for candidate to update itself
voteGranted	true means candidate received vote

Receiver implementation:

1. Reply false if term < currentTerm (§5.1)
2. If votedFor is null or candidateId, and candidate's log is at least as up-to-date as receiver's log, grant vote (§5.2, §5.4)

Rules for Servers

All Servers:

- If commitIndex > lastApplied: increment lastApplied, apply log[lastApplied] to state machine (§5.3)
- If RPC request or response contains term T > currentTerm: set currentTerm = T, convert to follower (§5.1)

Followers (§5.2):

- Respond to RPCs from candidates and leaders
- If election timeout elapses without receiving AppendEntries RPC from current leader or granting vote to candidate: convert to candidate

Candidates (§5.2):

- On conversion to candidate, start election:
  - Increment currentTerm
  - Vote for self
  - Reset election timer
  - Send RequestVote RPCs to all other servers
- If votes received from majority of servers: become leader
- If AppendEntries RPC received from new leader: convert to follower
- If election timeout elapses: start new election

Leaders:

- Upon election: send initial empty AppendEntries RPCs (heartbeat) to each server; repeat during idle periods to prevent election timeouts (§5.2)
- If command received from client: append entry to local log, respond after entry applied to state machine (§5.3)
- If last log index ≥ nextIndex for a follower: send AppendEntries RPC with log entries starting at nextIndex
  - If successful: update nextIndex and matchIndex for follower (§5.3)
  - If AppendEntries fails because of log inconsistency: decrement nextIndex and retry (§5.3)
- If there exists an N such that N > commitIndex, a majority of matchIndex[i] ≥ N, and log[N].term == currentTerm: set commitIndex = N (§5.3, §5.4).

Figure 2: A condensed summary of the Raft consensus algorithm (excluding membership changes and log compaction). The server behavior in the upper-left box is described as a set of rules that trigger independently and repeatedly. Section numbers such as §5.2 indicate where particular features are discussed. A formal specification [31] describes the algorithm more precisely.

Election Safety: at most one leader can be elected in a given term. §5.2

Leader Append-Only: a leader never overwrites or deletes entries in its log; it only appends new entries. §5.3

Log Matching: if two logs contain an entry with the same index and term, then the logs are identical in all entries up through the given index. §5.3

Leader Completeness: if a log entry is committed in a given term, then that entry will be present in the logs of the leaders for all higher-numbered terms. §5.4

State Machine Safety: if a server has applied a log entry at a given index to its state machine, no other server will ever apply a different log entry for the same index. §5.4.3

Figure 3: Raft guarantees that each of these properties is true at all times. The section numbers indicate where each property is discussed.

from clients and replicate them across the cluster, forcing the other logs to agree with its own (Section 5.3).

- **Safety:** the key safety property for Raft is the State Machine Safety Property in Figure 3: if any server has applied a particular log entry to its state machine, then no other server may apply a different command for the same log index. Section 5.4 describes how Raft ensures this property; the solution involves an additional restriction on the election mechanism described in Section 5.2.

After presenting the consensus algorithm, this section discusses the issue of availability and the role of timing in the system.

5.1 Raft basics

A Raft cluster contains several servers; five is a typical number, which allows the system to tolerate two failures. At any given time each server is in one of three states: *leader*, *follower*, or *candidate*. In normal operation there is exactly one leader and all of the other servers are followers. Followers are passive: they issue no requests on their own but simply respond to requests from leaders and candidates. The leader handles all client requests (if a client contacts a follower, the follower redirects it to the leader). The third state, candidate, is used to elect a new leader as described in Section 5.2. Figure 4 shows the states and their transitions; the transitions are discussed below.

Raft divides time into *terms* of arbitrary length, as shown in Figure 5. Terms are numbered with consecutive integers. Each term begins with an *election*, in which one or more candidates attempt to become leader as described in Section 5.2. If a candidate wins the election, then it serves as leader for the rest of the term. In some situations an election will result in a split vote. In this case the term will end with no leader; a new term (with a new election)

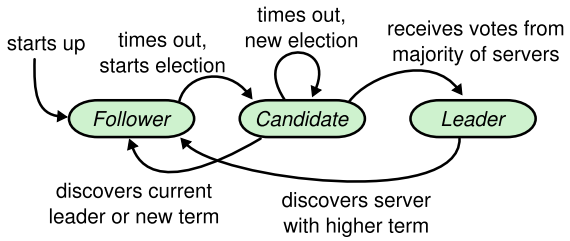


Figure 4: Server states. Followers only respond to requests from other servers. If a follower receives no communication, it becomes a candidate and initiates an election. A candidate that receives votes from a majority of the full cluster becomes the new leader. Leaders typically operate until they fail.

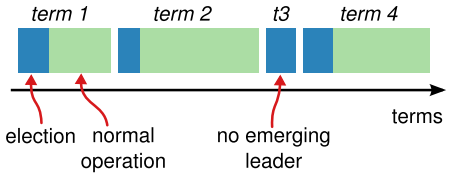


Figure 5: Time is divided into terms, and each term begins with an election. After a successful election, a single leader manages the cluster until the end of the term. Some elections fail, in which case the term ends without choosing a leader. The transitions between terms may be observed at different times on different servers.

will begin shortly. Raft ensures that there is at most one leader in a given term.

Different servers may observe the transitions between terms at different times, and in some situations a server may not observe an election or even entire terms. Terms act as a logical clock [14] in Raft, and they allow servers to detect obsolete information such as stale leaders. Each server stores a *current term* number, which increases monotonically over time. Current terms are exchanged whenever servers communicate; if one server's current term is smaller than the other's, then it updates its current term to the larger value. If a candidate or leader discovers that its term is out of date, it immediately reverts to follower state. If a server receives a request with a stale term number, it rejects the request.

Raft servers communicate using remote procedure calls (RPCs), and the basic consensus algorithm requires only two types of RPCs. RequestVote RPCs are initiated by candidates during elections (Section 5.2), and AppendEntries RPCs are initiated by leaders to replicate log entries and to provide a form of heartbeat (Section 5.3). Section 7 adds a third RPC for transferring snapshots between servers. Servers retry RPCs if they do not receive a response in a timely manner, and they issue RPCs in parallel for best performance.

5.2 Leader election

Raft uses a heartbeat mechanism to trigger leader election. When servers start up, they begin as followers. A server remains in follower state as long as it receives valid

RPCs from a leader or candidate. Leaders send periodic heartbeats (AppendEntries RPCs that carry no log entries) to all followers in order to maintain their authority. If a follower receives no communication over a period of time called the *election timeout*, then it assumes there is no viable leader and begins an election to choose a new leader.

To begin an election, a follower increments its current term and transitions to candidate state. It then votes for itself and issues RequestVote RPCs in parallel to each of the other servers in the cluster. A candidate continues in this state until one of three things happens: (a) it wins the election, (b) another server establishes itself as leader, or (c) a period of time goes by with no winner. These outcomes are discussed separately in the paragraphs below.

A candidate wins an election if it receives votes from a majority of the servers in the full cluster for the same term. Each server will vote for at most one candidate in a given term, on a first-come-first-served basis (note: Section 5.4 adds an additional restriction on votes). The majority rule ensures that at most one candidate can win the election for a particular term (the Election Safety Property in Figure 3). Once a candidate wins an election, it becomes leader. It then sends heartbeat messages to all of the other servers to establish its authority and prevent new elections.

While waiting for votes, a candidate may receive an AppendEntries RPC from another server claiming to be leader. If the leader's term (included in its RPC) is at least as large as the candidate's current term, then the candidate recognizes the leader as legitimate and returns to follower state. If the term in the RPC is smaller than the candidate's current term, then the candidate rejects the RPC and continues in candidate state.

The third possible outcome is that a candidate neither wins nor loses the election: if many followers become candidates at the same time, votes could be split so that no candidate obtains a majority. When this happens, each candidate will time out and start a new election by incrementing its term and initiating another round of RequestVote RPCs. However, without extra measures split votes could repeat indefinitely.

Raft uses randomized election timeouts to ensure that split votes are rare and that they are resolved quickly. To prevent split votes in the first place, election timeouts are chosen randomly from a fixed interval (e.g., 150–300ms). This spreads out the servers so that in most cases only a single server will time out; it wins the election and sends heartbeats before any other servers time out. The same mechanism is used to handle split votes. Each candidate restarts its randomized election timeout at the start of an election, and it waits for that timeout to elapse before starting the next election; this reduces the likelihood of another split vote in the new election. Section 9.3 shows that this approach elects a leader rapidly.

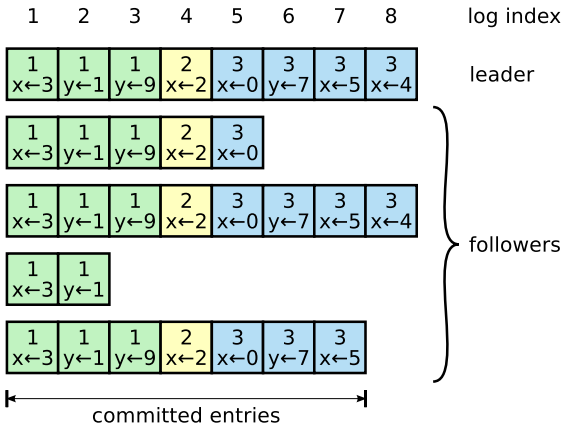


Figure 6: Logs are composed of entries, which are numbered sequentially. Each entry contains the term in which it was created (the number in each box) and a command for the state machine. An entry is considered *committed* if it is safe for that entry to be applied to state machines.

Elections are an example of how understandability guided our choice between design alternatives. Initially we planned to use a ranking system: each candidate was assigned a unique rank, which was used to select between competing candidates. If a candidate discovered another candidate with higher rank, it would return to follower state so that the higher ranking candidate could more easily win the next election. We found that this approach created subtle issues around availability (a lower-ranked server might need to time out and become a candidate again if a higher-ranked server fails, but if it does so too soon, it can reset progress towards electing a leader). We made adjustments to the algorithm several times, but after each adjustment new corner cases appeared. Eventually we concluded that the randomized retry approach is more obvious and understandable.

5.3 Log replication

Once a leader has been elected, it begins servicing client requests. Each client request contains a command to be executed by the replicated state machines. The leader appends the command to its log as a new entry, then issues AppendEntries RPCs in parallel to each of the other servers to replicate the entry. When the entry has been safely replicated (as described below), the leader applies the entry to its state machine and returns the result of that execution to the client. If followers crash or run slowly, or if network packets are lost, the leader retries AppendEntries RPCs indefinitely (even after it has responded to the client) until all followers eventually store all log entries.

Logs are organized as shown in Figure 6. Each log entry stores a state machine command along with the term number when the entry was received by the leader. The term numbers in log entries are used to detect inconsistencies between logs and to ensure some of the properties in Figure 3. Each log entry also has an integer index iden-

tifying its position in the log.

The leader decides when it is safe to apply a log entry to the state machines; such an entry is called *committed*. Raft guarantees that committed entries are durable and will eventually be executed by all of the available state machines. A log entry is committed once the leader that created the entry has replicated it on a majority of the servers (e.g., entry 7 in Figure 6). This also commits all preceding entries in the leader's log, including entries created by previous leaders. Section 5.4 discusses some subtleties when applying this rule after leader changes, and it also shows that this definition of commitment is safe. The leader keeps track of the highest index it knows to be committed, and it includes that index in future AppendEntries RPCs (including heartbeats) so that the other servers eventually find out. Once a follower learns that a log entry is committed, it applies the entry to its local state machine (in log order).

We designed the Raft log mechanism to maintain a high level of coherency between the logs on different servers. Not only does this simplify the system's behavior and make it more predictable, but it is an important component of ensuring safety. Raft maintains the following properties, which together constitute the Log Matching Property in Figure 3:

- If two entries in different logs have the same index and term, then they store the same command.
- If two entries in different logs have the same index and term, then the logs are identical in all preceding entries.

The first property follows from the fact that a leader creates at most one entry with a given log index in a given term, and log entries never change their position in the log. The second property is guaranteed by a simple consistency check performed by AppendEntries. When sending an AppendEntries RPC, the leader includes the index and term of the entry in its log that immediately precedes the new entries. If the follower does not find an entry in its log with the same index and term, then it refuses the new entries. The consistency check acts as an induction step: the initial empty state of the logs satisfies the Log Matching Property, and the consistency check preserves the Log Matching Property whenever logs are extended. As a result, whenever AppendEntries returns successfully, the leader knows that the follower's log is identical to its own log up through the new entries.

During normal operation, the logs of the leader and followers stay consistent, so the AppendEntries consistency check never fails. However, leader crashes can leave the logs inconsistent (the old leader may not have fully replicated all of the entries in its log). These inconsistencies can compound over a series of leader and follower crashes. Figure 7 illustrates the ways in which followers' logs may differ from that of a new leader. A follower may

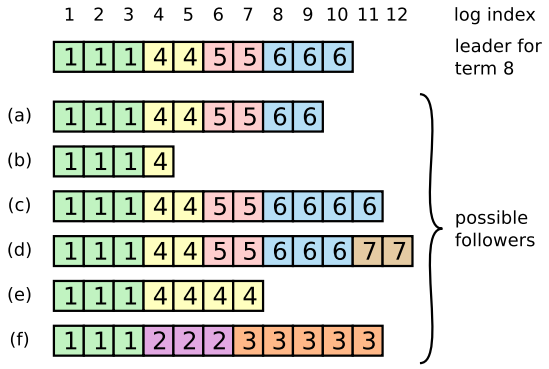


Figure 7: When the leader at the top comes to power, it is possible that any of scenarios (a–f) could occur in follower logs. Each box represents one log entry; the number in the box is its term. A follower may be missing entries (a–b), may have extra uncommitted entries (c–d), or both (e–f). For example, scenario (f) could occur if that server was the leader for term 2, added several entries to its log, then crashed before committing any of them; it restarted quickly, became leader for term 3, and added a few more entries to its log; before any of the entries in either term 2 or term 3 were committed, the server crashed again and remained down for several terms.

be missing entries that are present on the leader, it may have extra entries that are not present on the leader, or both. Missing and extraneous entries in a log may span multiple terms.

In Raft, the leader handles inconsistencies by forcing the followers' logs to duplicate its own. This means that conflicting entries in follower logs will be overwritten with entries from the leader's log. Section 5.4 will show that this is safe when coupled with one more restriction.

To bring a follower's log into consistency with its own, the leader must find the latest log entry where the two logs agree, delete any entries in the follower's log after that point, and send the follower all of the leader's entries after that point. All of these actions happen in response to the consistency check performed by AppendEntries RPCs. The leader maintains a *nextIndex* for each follower, which is the index of the next log entry the leader will send to that follower. When a leader first comes to power, it initializes all nextIndex values to the index just after the last one in its log (11 in Figure 7). If a follower's log is inconsistent with the leader's, the AppendEntries consistency check will fail in the next AppendEntries RPC. After a rejection, the leader decrements nextIndex and retries the AppendEntries RPC. Eventually nextIndex will reach a point where the leader and follower logs match. When this happens, AppendEntries will succeed, which removes any conflicting entries in the follower's log and appends entries from the leader's log (if any). Once AppendEntries succeeds, the follower's log is consistent with the leader's, and it will remain that way for the rest of the term.

If desired, the protocol can be optimized to reduce the number of rejected AppendEntries RPCs. For example, when rejecting an AppendEntries request, the follower

can include the term of the conflicting entry and the first index it stores for that term. With this information, the leader can decrement nextIndex to bypass all of the conflicting entries in that term; one AppendEntries RPC will be required for each term with conflicting entries, rather than one RPC per entry. In practice, we doubt this optimization is necessary, since failures happen infrequently and it is unlikely that there will be many inconsistent entries.

With this mechanism, a leader does not need to take any special actions to restore log consistency when it comes to power. It just begins normal operation, and the logs automatically converge in response to failures of the AppendEntries consistency check. A leader never overwrites or deletes entries in its own log (the Leader Append-Only Property in Figure 3).

This log replication mechanism exhibits the desirable consensus properties described in Section 2: Raft can accept, replicate, and apply new log entries as long as a majority of the servers are up; in the normal case a new entry can be replicated with a single round of RPCs to a majority of the cluster; and a single slow follower will not impact performance.

5.4 Safety

The previous sections described how Raft elects leaders and replicates log entries. However, the mechanisms described so far are not quite sufficient to ensure that each state machine executes exactly the same commands in the same order. For example, a follower might be unavailable while the leader commits several log entries, then it could be elected leader and overwrite these entries with new ones; as a result, different state machines might execute different command sequences.

This section completes the Raft algorithm by adding a restriction on which servers may be elected leader. The restriction ensures that the leader for any given term contains all of the entries committed in previous terms (the Leader Completeness Property from Figure 3). Given the election restriction, we then make the rules for commitment more precise. Finally, we present a proof sketch for the Leader Completeness Property and show how it leads to correct behavior of the replicated state machine.

5.4.1 Election restriction

In any leader-based consensus algorithm, the leader must eventually store all of the committed log entries. In some consensus algorithms, such as Viewstamped Replication [22], a leader can be elected even if it doesn't initially contain all of the committed entries. These algorithms contain additional mechanisms to identify the missing entries and transmit them to the new leader, either during the election process or shortly afterwards. Unfortunately, this results in considerable additional mechanism and complexity. Raft uses a simpler approach where it guarantees that all the committed entries from previous

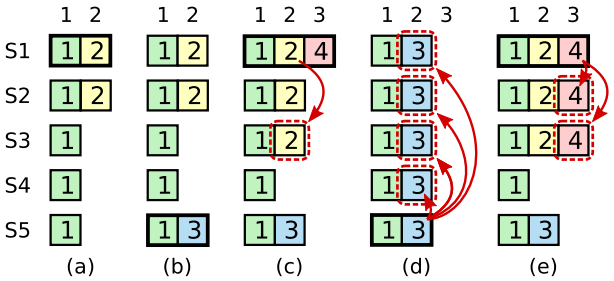


Figure 8: A time sequence showing why a leader cannot determine commitment using log entries from older terms. In (a) S1 is leader and partially replicates the log entry at index 2. In (b) S1 crashes; S5 is elected leader for term 3 with votes from S3, S4, and itself, and accepts a different entry at log index 2. In (c) S5 crashes; S1 restarts, is elected leader, and continues replication. At this point, the log entry from term 2 has been replicated on a majority of the servers, but it is not committed. If S1 crashes as in (d), S5 could be elected leader (with votes from S2, S3, and S4) and overwrite the entry with its own entry from term 3. However, if S1 replicates an entry from its current term on a majority of the servers before crashing, as in (e), then this entry is committed (S5 cannot win an election). At this point all preceding entries in the log are committed as well.

terms are present on each new leader from the moment of its election, without the need to transfer those entries to the leader. This means that log entries only flow in one direction, from leaders to followers, and leaders never overwrite existing entries in their logs.

Raft uses the voting process to prevent a candidate from winning an election unless its log contains all committed entries. A candidate must contact a majority of the cluster in order to be elected, which means that every committed entry must be present in at least one of those servers. If the candidate's log is at least as up-to-date as any other log in that majority (where "up-to-date" is defined precisely below), then it will hold all the committed entries. The RequestVote RPC implements this restriction: the RPC includes information about the candidate's log, and the voter denies its vote if its own log is more up-to-date than that of the candidate.

Raft determines which of two logs is more up-to-date by comparing the index and term of the last entries in the logs. If the logs have last entries with different terms, then the log with the later term is more up-to-date. If the logs end with the same term, then whichever log is longer is more up-to-date.

5.4.2 Committing entries from previous terms

As described in Section 5.3, a leader knows that an entry from its current term is committed once that entry is stored on a majority of the servers. If a leader crashes before committing an entry, future leaders will attempt to finish replicating the entry. However, a leader cannot immediately conclude that an entry from a previous term is committed once it is stored on a majority of servers. Fig-

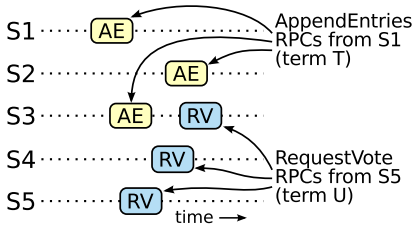


Figure 9: If S1 (leader for term T) commits a new log entry from its term, and S5 is elected leader for a later term U, then there must be at least one server (S3) that accepted the log entry and also voted for S5.

ure 8 illustrates a situation where an old log entry is stored on a majority of servers, yet can still be overwritten by a future leader.

To eliminate problems like the one in Figure 8, Raft never commits log entries from previous terms by counting replicas. Only log entries from the leader's current term are committed by counting replicas; once an entry from the current term has been committed in this way, then all prior entries are committed indirectly because of the Log Matching Property. There are some situations where a leader could safely conclude that an older log entry is committed (for example, if that entry is stored on every server), but Raft takes a more conservative approach for simplicity.

Raft incurs this extra complexity in the commitment rules because log entries retain their original term numbers when a leader replicates entries from previous terms. In other consensus algorithms, if a new leader re-replicates entries from prior "terms," it must do so with its new "term number." Raft's approach makes it easier to reason about log entries, since they maintain the same term number over time and across logs. In addition, new leaders in Raft send fewer log entries from previous terms than in other algorithms (other algorithms must send redundant log entries to renumber them before they can be committed).

5.4.3 Safety argument

Given the complete Raft algorithm, we can now argue more precisely that the Leader Completeness Property holds (this argument is based on the safety proof; see Section 9.2). We assume that the Leader Completeness Property does not hold, then we prove a contradiction. Suppose the leader for term T (leader_T) commits a log entry from its term, but that log entry is not stored by the leader of some future term. Consider the smallest term U $>$ T whose leader (leader_U) does not store the entry.

1. The committed entry must have been absent from leader_U's log at the time of its election (leaders never delete or overwrite entries).
2. leader_T replicated the entry on a majority of the cluster, and leader_U received votes from a majority of the cluster. Thus, at least one server ("the voter") both accepted the entry from leader_T and voted for leader_U, as shown in Figure 9. The voter is key to reaching a contradiction.
3. The voter must have accepted the committed entry from leader_T *before* voting for leader_U; otherwise it would have rejected the AppendEntries request from leader_T (its current term would have been higher than T).
4. The voter still stored the entry when it voted for leader_U, since every intervening leader contained the entry (by assumption), leaders never remove entries, and followers only remove entries if they conflict with the leader.
5. The voter granted its vote to leader_U, so leader_U's log must have been as up-to-date as the voter's. This leads to one of two contradictions.
6. First, if the voter and leader_U shared the same last log term, then leader_U's log must have been at least as long as the voter's, so its log contained every entry in the voter's log. This is a contradiction, since the voter contained the committed entry and leader_U was assumed not to.
7. Otherwise, leader_U's last log term must have been larger than the voter's. Moreover, it was larger than T, since the voter's last log term was at least T (it contains the committed entry from term T). The earlier leader that created leader_U's last log entry must have contained the committed entry in its log (by assumption). Then, by the Log Matching Property, leader_U's log must also contain the committed entry, which is a contradiction.
8. This completes the contradiction. Thus, the leaders of all terms greater than T must contain all entries from term T that are committed in term T.
9. The Log Matching Property guarantees that future leaders will also contain entries that are committed indirectly, such as index 2 in Figure 8(d).

Given the Leader Completeness Property, we can prove the State Machine Safety Property from Figure 3, which states that if a server has applied a log entry at a given index to its state machine, no other server will ever apply a different log entry for the same index. At the time a server applies a log entry to its state machine, its log must be identical to the leader's log up through that entry and the entry must be committed. Now consider the lowest term in which any server applies a given log index; the Log Completeness Property guarantees that the leaders for all higher terms will store that same log entry, so servers that apply the index in later terms will apply the same value. Thus, the State Machine Safety Property holds.

Finally, Raft requires servers to apply entries in log index order. Combined with the State Machine Safety Property, this means that all servers will apply exactly the same set of log entries to their state machines, in the same order.

5.5 Follower and candidate crashes

Until this point we have focused on leader failures. Follower and candidate crashes are much simpler to handle than leader crashes, and they are both handled in the same way. If a follower or candidate crashes, then future RequestVote and AppendEntries RPCs sent to it will fail. Raft handles these failures by retrying indefinitely; if the crashed server restarts, then the RPC will complete successfully. If a server crashes after completing an RPC but before responding, then it will receive the same RPC again after it restarts. Raft RPCs are idempotent, so this causes no harm. For example, if a follower receives an AppendEntries request that includes log entries already present in its log, it ignores those entries in the new request.

5.6 Timing and availability

One of our requirements for Raft is that safety must not depend on timing: the system must not produce incorrect results just because some event happens more quickly or slowly than expected. However, availability (the ability of the system to respond to clients in a timely manner) must inevitably depend on timing. For example, if message exchanges take longer than the typical time between server crashes, candidates will not stay up long enough to win an election; without a steady leader, Raft cannot make progress.

Leader election is the aspect of Raft where timing is most critical. Raft will be able to elect and maintain a steady leader as long as the system satisfies the following *timing requirement*:

$$broadcastTime \ll electionTimeout \ll MTBF$$

In this inequality *broadcastTime* is the average time it takes a server to send RPCs in parallel to every server in the cluster and receive their responses; *electionTimeout* is the election timeout described in Section 5.2; and *MTBF* is the average time between failures for a single server. The broadcast time should be an order of magnitude less than the election timeout so that leaders can reliably send the heartbeat messages required to keep followers from starting elections; given the randomized approach used for election timeouts, this inequality also makes split votes unlikely. The election timeout should be a few orders of magnitude less than MTBF so that the system makes steady progress. When the leader crashes, the system will be unavailable for roughly the election timeout; we would like this to represent only a small fraction of overall time.

The broadcast time and MTBF are properties of the underlying system, while the election timeout is something we must choose. Raft's RPCs typically require the recipient to persist information to stable storage, so the broadcast time may range from 0.5ms to 20ms, depending on storage technology. As a result, the election timeout is likely to be somewhere between 10ms and 500ms. Typical server MTBFs are several months or more, which easily satisfies the timing requirement.

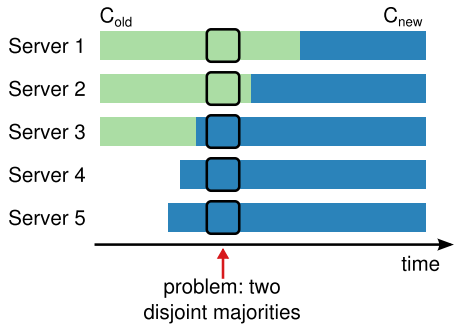


Figure 10: Switching directly from one configuration to another is unsafe because different servers will switch at different times. In this example, the cluster grows from three servers to five. Unfortunately, there is a point in time where two different leaders can be elected for the same term, one with a majority of the old configuration (C_{old}) and another with a majority of the new configuration (C_{new}).

6 Cluster membership changes

Up until now we have assumed that the cluster *configuration* (the set of servers participating in the consensus algorithm) is fixed. In practice, it will occasionally be necessary to change the configuration, for example to replace servers when they fail or to change the degree of replication. Although this can be done by taking the entire cluster off-line, updating configuration files, and then restarting the cluster, this would leave the cluster unavailable during the changeover. In addition, if there are any manual steps, they risk operator error. In order to avoid these issues, we decided to automate configuration changes and incorporate them into the Raft consensus algorithm.

For the configuration change mechanism to be safe, there must be no point during the transition where it is possible for two leaders to be elected for the same term. Unfortunately, any approach where servers switch directly from the old configuration to the new configuration is unsafe. It isn't possible to atomically switch all of the servers at once, so the cluster can potentially split into two independent majorities during the transition (see Figure 10).

In order to ensure safety, configuration changes must use a two-phase approach. There are a variety of ways to implement the two phases. For example, some systems (e.g., [22]) use the first phase to disable the old configuration so it cannot process client requests; then the second phase enables the new configuration. In Raft the cluster first switches to a transitional configuration we call *joint consensus*; once the joint consensus has been committed, the system then transitions to the new configuration. The joint consensus combines both the old and new configurations:

- Log entries are replicated to all servers in both configurations.

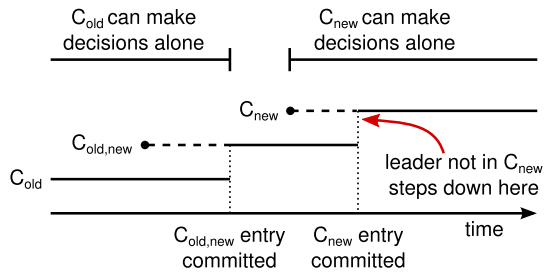


Figure 11: Timeline for a configuration change. Dashed lines show configuration entries that have been created but not committed, and solid lines show the latest committed configuration entry. The leader first creates the $C_{old,new}$ configuration entry in its log and commits it to $C_{old,new}$ (a majority of C_{old} and a majority of C_{new}). Then it creates the C_{new} entry and commits it to a majority of C_{new}. There is no point in time in which C_{old} and C_{new} can both make decisions independently.

- Any server from either configuration may serve as leader.
- Agreement (for elections and entry commitment) requires separate majorities from *both* the old and new configurations.

The joint consensus allows individual servers to transition between configurations at different times without compromising safety. Furthermore, joint consensus allows the cluster to continue servicing client requests throughout the configuration change.

Cluster configurations are stored and communicated using special entries in the replicated log; Figure 11 illustrates the configuration change process. When the leader receives a request to change the configuration from C_{old} to C_{new}, it stores the configuration for joint consensus ($C_{old,new}$ in the figure) as a log entry and replicates that entry using the mechanisms described previously. Once a given server adds the new configuration entry to its log, it uses that configuration for all future decisions (a server always uses the latest configuration in its log, regardless of whether the entry is committed). This means that the leader will use the rules of $C_{old,new}$ to determine when the log entry for $C_{old,new}$ is committed. If the leader crashes, a new leader may be chosen under either C_{old} or $C_{old,new}$, depending on whether the winning candidate has received $C_{old,new}$. In any case, C_{new} cannot make unilateral decisions during this period.

Once $C_{old,new}$ has been committed, neither C_{old} nor C_{new} can make decisions without approval of the other, and the Leader Completeness Property ensures that only servers with the $C_{old,new}$ log entry can be elected as leader. It is now safe for the leader to create a log entry describing C_{new} and replicate it to the cluster. Again, this configuration will take effect on each server as soon as it is seen. When the new configuration has been committed under the rules of C_{new}, the old configuration is irrelevant and servers not in the new configuration can be shut down. As shown in Figure 11, there is no time when C_{old} and C_{new} can both make unilateral decisions; this guarantees safety.

There are three more issues to address for reconfiguration. The first issue is that new servers may not initially store any log entries. If they are added to the cluster in this state, it could take quite a while for them to catch up, during which time it might not be possible to commit new log entries. In order to avoid availability gaps, Raft introduces an additional phase before the configuration change, in which the new servers join the cluster as non-voting members (the leader replicates log entries to them, but they are not considered for majorities). Once the new servers have caught up with the rest of the cluster, the reconfiguration can proceed as described above.

The second issue is that the cluster leader may not be part of the new configuration. In this case, the leader steps down (returns to follower state) once it has committed the C_{new} log entry. This means that there will be a period of time (while it is committing C_{new}) when the leader is managing a cluster that does not include itself; it replicates log entries but does not count itself in majorities. The leader transition occurs when C_{new} is committed because this is the first point when the new configuration can operate independently (it will always be possible to choose a leader from C_{new}). Before this point, it may be the case that only a server from C_{old} can be elected leader.

The third issue is that removed servers (those not in C_{new}) can disrupt the cluster. These servers will not receive heartbeats, so they will time out and start new elections. They will then send RequestVote RPCs with new term numbers, and this will cause the current leader to revert to follower state. A new leader will eventually be elected, but the removed servers will time out again and the process will repeat, resulting in poor availability.

To prevent this problem, servers disregard RequestVote RPCs when they believe a current leader exists. Specifically, if a server receives a RequestVote RPC within the minimum election timeout of hearing from a current leader, it does not update its term or grant its vote. This does not affect normal elections, where each server waits at least a minimum election timeout before starting an election. However, it helps avoid disruptions from removed servers: if a leader is able to get heartbeats to its cluster, then it will not be deposed by larger term numbers.

7 Log compaction

Raft's log grows during normal operation to incorporate more client requests, but in a practical system, it cannot grow without bound. As the log grows longer, it occupies more space and takes more time to replay. This will eventually cause availability problems without some mechanism to discard obsolete information that has accumulated in the log.

Snapshotting is the simplest approach to compaction. In snapshotting, the entire current system state is written to a *snapshot* on stable storage, then the entire log up to

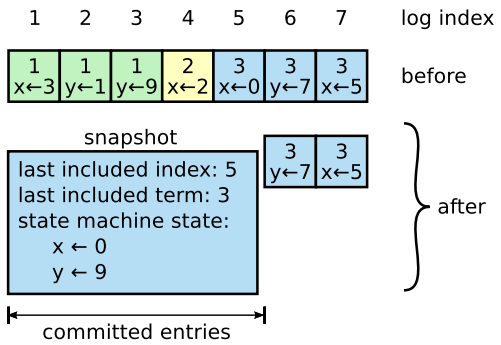


Figure 12: A server replaces the committed entries in its log (indexes 1 through 5) with a new snapshot, which stores just the current state (variables *x* and *y* in this example). The snapshot's last included index and term serve to position the snapshot in the log preceding entry 6.

that point is discarded. Snapshotting is used in Chubby and ZooKeeper, and the remainder of this section describes snapshotting in Raft.

Incremental approaches to compaction, such as log cleaning [36] and log-structured merge trees [30, 5], are also possible. These operate on a fraction of the data at once, so they spread the load of compaction more evenly over time. They first select a region of data that has accumulated many deleted and overwritten objects, then they rewrite the live objects from that region more compactly and free the region. This requires significant additional mechanism and complexity compared to snapshotting, which simplifies the problem by always operating on the entire data set. While log cleaning would require modifications to Raft, state machines can implement LSM trees using the same interface as snapshotting.

Figure 12 shows the basic idea of snapshotting in Raft. Each server takes snapshots independently, covering just the committed entries in its log. Most of the work consists of the state machine writing its current state to the snapshot. Raft also includes a small amount of metadata in the snapshot: the *last included index* is the index of the last entry in the log that the snapshot replaces (the last entry the state machine had applied), and the *last included term* is the term of this entry. These are preserved to support the AppendEntries consistency check for the first log entry following the snapshot, since that entry needs a previous log index and term. To enable cluster membership changes (Section 6), the snapshot also includes the latest configuration in the log as of last included index. Once a server completes writing a snapshot, it may delete all log entries up through the last included index, as well as any prior snapshot.

Although servers normally take snapshots independently, the leader must occasionally send snapshots to followers that lag behind. This happens when the leader has already discarded the next log entry that it needs to send to a follower. Fortunately, this situation is unlikely in normal operation: a follower that has kept up with the

InstallSnapshot RPC

Invoked by leader to send chunks of a snapshot to a follower. Leaders always send chunks in order.

Arguments:

term	leader's term
leaderId	so follower can redirect clients
lastIncludedIndex	the snapshot replaces all entries up through and including this index
lastIncludedTerm	term of lastIncludedIndex
offset	byte offset where chunk is positioned in the snapshot file
data[]	raw bytes of the snapshot chunk, starting at offset
done	true if this is the last chunk

Results:

term	currentTerm, for leader to update itself

Receiver implementation:

1. Reply immediately if term < currentTerm
2. Create new snapshot file if first chunk (offset is 0)
3. Write data into snapshot file at given offset
4. Reply and wait for more data chunks if done is false
5. Save snapshot file, discard any existing or partial snapshot with a smaller index
6. If existing log entry has same index and term as snapshot's last included entry, retain log entries following it and reply
7. Discard the entire log
8. Reset state machine using snapshot contents (and load snapshot's cluster configuration)

Figure 13: A summary of the InstallSnapshot RPC. Snapshots are split into chunks for transmission; this gives the follower a sign of life with each chunk, so it can reset its election timer.

leader would already have this entry. However, an exceptionally slow follower or a new server joining the cluster (Section 6) would not. The way to bring such a follower up-to-date is for the leader to send it a snapshot over the network.

The leader uses a new RPC called InstallSnapshot to send snapshots to followers that are too far behind; see Figure 13. When a follower receives a snapshot with this RPC, it must decide what to do with its existing log entries. Usually the snapshot will contain new information not already in the recipient's log. In this case, the follower discards its entire log; it is all superseded by the snapshot and may possibly have uncommitted entries that conflict with the snapshot. If instead the follower receives a snapshot that describes a prefix of its log (due to retransmission or by mistake), then log entries covered by the snapshot are deleted but entries following the snapshot are still valid and must be retained.

This snapshotting approach departs from Raft's strong leader principle, since followers can take snapshots without the knowledge of the leader. However, we think this departure is justified. While having a leader helps avoid conflicting decisions in reaching consensus, consensus has already been reached when snapshotting, so no decisions conflict. Data still only flows from leaders to fol-

lowers, just followers can now reorganize their data.

We considered an alternative leader-based approach in which only the leader would create a snapshot, then it would send this snapshot to each of its followers. However, this has two disadvantages. First, sending the snapshot to each follower would waste network bandwidth and slow the snapshotting process. Each follower already has the information needed to produce its own snapshots, and it is typically much cheaper for a server to produce a snapshot from its local state than it is to send and receive one over the network. Second, the leader's implementation would be more complex. For example, the leader would need to send snapshots to followers in parallel with replicating new log entries to them, so as not to block new client requests.

There are two more issues that impact snapshotting performance. First, servers must decide when to snapshot. If a server snapshots too often, it wastes disk bandwidth and energy; if it snapshots too infrequently, it risks exhausting its storage capacity, and it increases the time required to replay the log during restarts. One simple strategy is to take a snapshot when the log reaches a fixed size in bytes. If this size is set to be significantly larger than the expected size of a snapshot, then the disk bandwidth overhead for snapshotting will be small.

The second performance issue is that writing a snapshot can take a significant amount of time, and we do not want this to delay normal operations. The solution is to use copy-on-write techniques so that new updates can be accepted without impacting the snapshot being written. For example, state machines built with functional data structures naturally support this. Alternatively, the operating system's copy-on-write support (e.g., fork on Linux) can be used to create an in-memory snapshot of the entire state machine (our implementation uses this approach).

8 Client interaction

This section describes how clients interact with Raft, including how clients find the cluster leader and how Raft supports linearizable semantics [10]. These issues apply to all consensus-based systems, and Raft's solutions are similar to other systems.

Clients of Raft send all of their requests to the leader. When a client first starts up, it connects to a randomly-chosen server. If the client's first choice is not the leader, that server will reject the client's request and supply information about the most recent leader it has heard from (AppendEntries requests include the network address of the leader). If the leader crashes, client requests will time out; clients then try again with randomly-chosen servers.

Our goal for Raft is to implement linearizable semantics (each operation appears to execute instantaneously, exactly once, at some point between its invocation and its response). However, as described so far Raft can execute a command multiple times: for example, if the leader crashes after committing the log entry but before responding to the client, the client will retry the command with a new leader, causing it to be executed a second time. The solution is for clients to assign unique serial numbers to every command. Then, the state machine tracks the latest serial number processed for each client, along with the associated response. If it receives a command whose serial number has already been executed, it responds immediately without re-executing the request.

Read-only operations can be handled without writing anything into the log. However, with no additional measures, this would run the risk of returning stale data, since the leader responding to the request might have been superseded by a newer leader of which it is unaware. Linearizable reads must not return stale data, and Raft needs two extra precautions to guarantee this without using the log. First, a leader must have the latest information on which entries are committed. The Leader Completeness Property guarantees that a leader has all committed entries, but at the start of its term, it may not know which those are. To find out, it needs to commit an entry from its term. Raft handles this by having each leader commit a blank *no-op* entry into the log at the start of its term. Second, a leader must check whether it has been deposed before processing a read-only request (its information may be stale if a more recent leader has been elected). Raft handles this by having the leader exchange heartbeat messages with a majority of the cluster before responding to read-only requests. Alternatively, the leader could rely on the heartbeat mechanism to provide a form of lease [9], but this would rely on timing for safety (it assumes bounded clock skew).

9 Implementation and evaluation

We have implemented Raft as part of a replicated state machine that stores configuration information for RAMCloud [33] and assists in failover of the RAMCloud coordinator. The Raft implementation contains roughly 2000 lines of C++ code, not including tests, comments, or blank lines. The source code is freely available [23]. There are also about 25 independent third-party open source implementations [34] of Raft in various stages of development, based on drafts of this paper. Also, various companies are deploying Raft-based systems [34].

The remainder of this section evaluates Raft using three criteria: understandability, correctness, and performance.

9.1 Understandability

To measure Raft's understandability relative to Paxos, we conducted an experimental study using upper-level undergraduate and graduate students in an Advanced Operating Systems course at Stanford University and a Distributed Computing course at U.C. Berkeley. We recorded a video lecture of Raft and another of Paxos, and created corresponding quizzes. The Raft lecture covered the content of this paper except for log compaction; the Paxos

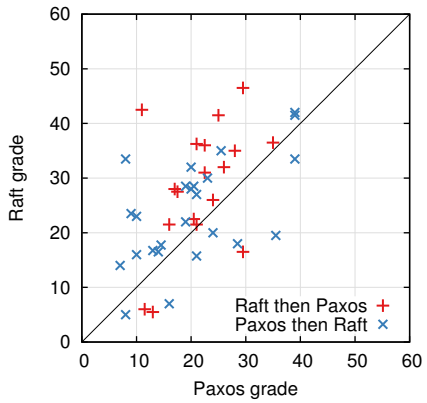


Figure 14: A scatter plot comparing 43 participants' performance on the Raft and Paxos quizzes. Points above the diagonal (33) represent participants who scored higher for Raft.

lecture covered enough material to create an equivalent replicated state machine, including single-decree Paxos, multi-decree Paxos, reconfiguration, and a few optimizations needed in practice (such as leader election). The quizzes tested basic understanding of the algorithms and also required students to reason about corner cases. Each student watched one video, took the corresponding quiz, watched the second video, and took the second quiz. About half of the participants did the Paxos portion first and the other half did the Raft portion first in order to account for both individual differences in performance and experience gained from the first portion of the study. We compared participants' scores on each quiz to determine whether participants showed a better understanding of Raft.

We tried to make the comparison between Paxos and Raft as fair as possible. The experiment favored Paxos in two ways: 15 of the 43 participants reported having some prior experience with Paxos, and the Paxos video is 14% longer than the Raft video. As summarized in Table 1, we have taken steps to mitigate potential sources of bias. All of our materials are available for review [28, 31].

On average, participants scored 4.9 points higher on the Raft quiz than on the Paxos quiz (out of a possible 60 points, the mean Raft score was 25.7 and the mean Paxos score was 20.8); Figure 14 shows their individual scores. A paired *t*-test states that, with 95% confidence, the true distribution of Raft scores has a mean at least 2.5 points larger than the true distribution of Paxos scores.

We also created a linear regression model that predicts a new student's quiz scores based on three factors: which quiz they took, their degree of prior Paxos experience, and

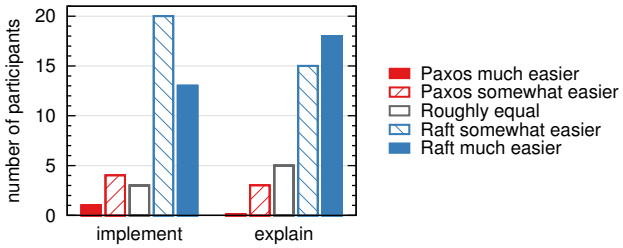


Figure 15: Using a 5-point scale, participants were asked (left) which algorithm they felt would be easier to implement in a functioning, correct, and efficient system, and (right) which would be easier to explain to a CS graduate student.

the order in which they learned the algorithms. The model predicts that the choice of quiz produces a 12.5-point difference in favor of Raft. This is significantly higher than the observed difference of 4.9 points, because many of the actual students had prior Paxos experience, which helped Paxos considerably, whereas it helped Raft slightly less. Curiously, the model also predicts scores 6.3 points lower on Raft for people that have already taken the Paxos quiz; although we don't know why, this does appear to be statistically significant.

We also surveyed participants after their quizzes to see which algorithm they felt would be easier to implement or explain; these results are shown in Figure 15. An overwhelming majority of participants reported Raft would be easier to implement and explain (33 of 41 for each question). However, these self-reported feelings may be less reliable than participants' quiz scores, and participants may have been biased by knowledge of our hypothesis that Raft is easier to understand.

A detailed discussion of the Raft user study is available at [31].

9.2 Correctness

We have developed a formal specification and a proof of safety for the consensus mechanism described in Section 5. The formal specification [31] makes the information summarized in Figure 2 completely precise using the TLA+ specification language [17]. It is about 400 lines long and serves as the subject of the proof. It is also useful on its own for anyone implementing Raft. We have mechanically proven the Log Completeness Property using the TLA proof system [7]. However, this proof relies on invariants that have not been mechanically checked (for example, we have not proven the type safety of the specification). Furthermore, we have written an informal proof [31] of the State Machine Safety property which is complete (it relies on the specification alone) and rela-

Concern	Steps taken to mitigate bias	Materials for review [28, 31]
Equal lecture quality	Same lecturer for both. Paxos lecture based on and improved from existing materials used in several universities. Paxos lecture is 14% longer.	videos
Equal quiz difficulty	Questions grouped in difficulty and paired across exams.	quizzes
Fair grading	Used rubric. Graded in random order, alternating between quizzes.	rubric

Table 1: Concerns of possible bias against Paxos in the study, steps taken to counter each, and additional materials available.

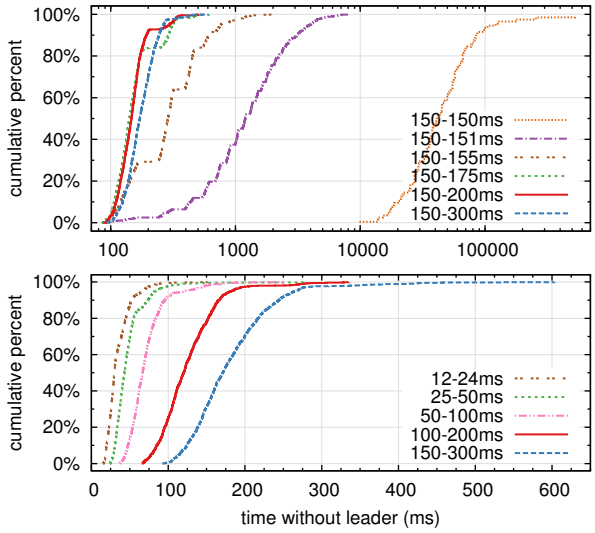


Figure 16: The time to detect and replace a crashed leader. The top graph varies the amount of randomness in election timeouts, and the bottom graph scales the minimum election timeout. Each line represents 1000 trials (except for 100 trials for "150–150ms") and corresponds to a particular choice of election timeouts; for example, "150–155ms" means that election timeouts were chosen randomly and uniformly between 150ms and 155ms. The measurements were taken on a cluster of five servers with a broadcast time of roughly 15ms. Results for a cluster of nine servers are similar.

tively precise (it is about 3500 words long).

9.3 Performance

Raft's performance is similar to other consensus algorithms such as Paxos. The most important case for performance is when an established leader is replicating new log entries. Raft achieves this using the minimal number of messages (a single round-trip from the leader to half the cluster). It is also possible to further improve Raft's performance. For example, it easily supports batching and pipelining requests for higher throughput and lower latency. Various optimizations have been proposed in the literature for other algorithms; many of these could be applied to Raft, but we leave this to future work.

We used our Raft implementation to measure the performance of Raft's leader election algorithm and answer two questions. First, does the election process converge quickly? Second, what is the minimum downtime that can be achieved after leader crashes?

To measure leader election, we repeatedly crashed the leader of a cluster of five servers and timed how long it took to detect the crash and elect a new leader (see Figure 16). To generate a worst-case scenario, the servers in each trial had different log lengths, so some candidates were not eligible to become leader. Furthermore, to encourage split votes, our test script triggered a synchronized broadcast of heartbeat RPCs from the leader before terminating its process (this approximates the behavior of the leader replicating a new log entry prior to crashing). The leader was crashed uniformly randomly within its heartbeat interval, which was half of the minimum election timeout for all tests. Thus, the smallest possible downtime was about half of the minimum election timeout.

The top graph in Figure 16 shows that a small amount of randomization in the election timeout is enough to avoid split votes in elections. In the absence of randomness, leader election consistently took longer than 10 seconds in our tests due to many split votes. Adding just 5ms of randomness helps significantly, resulting in a median downtime of 287ms. Using more randomness improves worst-case behavior: with 50ms of randomness the worst-case completion time (over 1000 trials) was 513ms.

The bottom graph in Figure 16 shows that downtime can be reduced by reducing the election timeout. With an election timeout of 12–24ms, it takes only 35ms on average to elect a leader (the longest trial took 152ms). However, lowering the timeouts beyond this point violates Raft's timing requirement: leaders have difficulty broadcasting heartbeats before other servers start new elections. This can cause unnecessary leader changes and lower overall system availability. We recommend using a conservative election timeout such as 150–300ms; such timeouts are unlikely to cause unnecessary leader changes and will still provide good availability.

10 Related work

There have been numerous publications related to consensus algorithms, many of which fall into one of the following categories:

- Lamport's original description of Paxos [15], and attempts to explain it more clearly [16, 20, 21].
- Elaborations of Paxos, which fill in missing details and modify the algorithm to provide a better foundation for implementation [26, 39, 13].
- Systems that implement consensus algorithms, such as Chubby [2, 4], ZooKeeper [11, 12], and Spanner [6]. The algorithms for Chubby and Spanner have not been published in detail, though both claim to be based on Paxos. ZooKeeper's algorithm has been published in more detail, but it is quite different from Paxos.
- Performance optimizations that can be applied to Paxos [18, 19, 3, 25, 1, 27].
- Oki and Liskov's Viewstamped Replication (VR), an alternative approach to consensus developed around the same time as Paxos. The original description [29] was intertwined with a protocol for distributed transactions, but the core consensus protocol has been separated in a recent update [22]. VR uses a leader-based approach with many similarities to Raft.

The greatest difference between Raft and Paxos is Raft's strong leadership: Raft uses leader election as an essential part of the consensus protocol, and it concen-

trates as much functionality as possible in the leader. This approach results in a simpler algorithm that is easier to understand. For example, in Paxos, leader election is orthogonal to the basic consensus protocol: it serves only as a performance optimization and is not required for achieving consensus. However, this results in additional mechanism: Paxos includes both a two-phase protocol for basic consensus and a separate mechanism for leader election. In contrast, Raft incorporates leader election directly into the consensus algorithm and uses it as the first of the two phases of consensus. This results in less mechanism than in Paxos.

Like Raft, VR and ZooKeeper are leader-based and therefore share many of Raft's advantages over Paxos. However, Raft has less mechanism that VR or ZooKeeper because it minimizes the functionality in non-leaders. For example, log entries in Raft flow in only one direction: outward from the leader in AppendEntries RPCs. In VR log entries flow in both directions (leaders can receive log entries during the election process); this results in additional mechanism and complexity. The published description of ZooKeeper also transfers log entries both to and from the leader, but the implementation is apparently more like Raft [35].

Raft has fewer message types than any other algorithm for consensus-based log replication that we are aware of. For example, we counted the message types VR and ZooKeeper use for basic consensus and membership changes (excluding log compaction and client interaction, as these are nearly independent of the algorithms). VR and ZooKeeper each define 10 different message types, while Raft has only 4 message types (two RPC requests and their responses). Raft's messages are a bit more dense than the other algorithms', but they are simpler collectively. In addition, VR and ZooKeeper are described in terms of transmitting entire logs during leader changes; additional message types will be required to optimize these mechanisms so that they are practical.

Raft's strong leadership approach simplifies the algorithm, but it precludes some performance optimizations. For example, Egalitarian Paxos (EPaxos) can achieve higher performance under some conditions with a leaderless approach [27]. EPaxos exploits commutativity in state machine commands. Any server can commit a command with just one round of communication as long as other commands that are proposed concurrently commute with it. However, if commands that are proposed concurrently do not commute with each other, EPaxos requires an additional round of communication. Because any server may commit commands, EPaxos balances load well between servers and is able to achieve lower latency than Raft in WAN settings. However, it adds significant complexity to Paxos.

Several different approaches for cluster membership changes have been proposed or implemented in other work, including Lamport's original proposal [15], VR [22], and SMART [24]. We chose the joint consensus approach for Raft because it leverages the rest of the consensus protocol, so that very little additional mechanism is required for membership changes. Lamport's α-based approach was not an option for Raft because it assumes consensus can be reached without a leader. In comparison to VR and SMART, Raft's reconfiguration algorithm has the advantage that membership changes can occur without limiting the processing of normal requests; in contrast, VR stops all normal processing during configuration changes, and SMART imposes an α-like limit on the number of outstanding requests. Raft's approach also adds less mechanism than either VR or SMART.

11 Conclusion

Algorithms are often designed with correctness, efficiency, and/or conciseness as the primary goals. Although these are all worthy goals, we believe that understandability is just as important. None of the other goals can be achieved until developers render the algorithm into a practical implementation, which will inevitably deviate from and expand upon the published form. Unless developers have a deep understanding of the algorithm and can create intuitions about it, it will be difficult for them to retain its desirable properties in their implementation.

In this paper we addressed the issue of distributed consensus, where a widely accepted but impenetrable algorithm, Paxos, has challenged students and developers for many years. We developed a new algorithm, Raft, which we have shown to be more understandable than Paxos. We also believe that Raft provides a better foundation for system building. Using understandability as the primary design goal changed the way we approached the design of Raft; as the design progressed we found ourselves reusing a few techniques repeatedly, such as decomposing the problem and simplifying the state space. These techniques not only improved the understandability of Raft but also made it easier to convince ourselves of its correctness.

12 Acknowledgments

The user study would not have been possible without the support of Ali Ghodsi, David Mazières, and the students of CS 294-91 at Berkeley and CS 240 at Stanford. Scott Klemmer helped us design the user study, and Nelson Ray advised us on statistical analysis. The Paxos slides for the user study borrowed heavily from a slide deck originally created by Lorenzo Alvisi. Special thanks go to David Mazières and Ezra Hoch for finding subtle bugs in Raft. Many people provided helpful feedback on the paper and user study materials, including Ed Bugnion, Michael Chan, Hugues Evrard,

Daniel Giffin, Arjun Gopalan, Jon Howell, Vimalkumar Jeyakumar, Ankita Kejriwal, Aleksandar Kracun, Amit Levy, Joel Martin, Satoshi Matsushita, Oleg Pesok, David Ramos, Robbert van Renesse, Mendel Rosenblum, Nicolas Schiper, Deian Stefan, Andrew Stone, Ryan Stutsman, David Terei, Stephen Yang, Matei Zaharia, 24 anonymous conference reviewers (with duplicates), and especially our shepherd Eddie Kohler. Werner Vogels tweeted a link to an earlier draft, which gave Raft significant exposure. This work was supported by the Gigascale Systems Research Center and the Multiscale Systems Center, two of six research centers funded under the Focus Center Research Program, a Semiconductor Research Corporation program, by STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA, by the National Science Foundation under Grant No. 0963859, and by grants from Facebook, Google, Mellanox, NEC, NetApp, SAP, and Samsung. Diego Ongaro is supported by The Junglee Corporation Stanford Graduate Fellowship.

References

[1] BOLOSKY, W. J., BRADSHAW, D., HAAGENS, R. B., KUSTERS, N. P., AND LI, P. Paxos replicated state machines as the basis of a high-performance data store. In *Proc. NSDI'11, USENIX Conference on Networked Systems Design and Implementation* (2011), USENIX, pp. 141–154.

[2] BURROWS, M. The Chubby lock service for loosely-coupled distributed systems. In *Proc. OSDI'06, Symposium on Operating Systems Design and Implementation* (2006), USENIX, pp. 335–350.

[3] CAMARGOS, L. J., SCHMIDT, R. M., AND PEDONE, F. Multicoordinated Paxos. In *Proc. PODC'07, ACM Symposium on Principles of Distributed Computing* (2007), ACM, pp. 316–317.

[4] CHANDRA, T. D., GRIESEMER, R., AND REDSTONE, J. Paxos made live: an engineering perspective. In *Proc. PODC'07, ACM Symposium on Principles of Distributed Computing* (2007), ACM, pp. 398–407.

[5] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. E. Bigtable: a distributed storage system for structured data. In *Proc. OSDI'06, USENIX Symposium on Operating Systems Design and Implementation* (2006), USENIX, pp. 205–218.

[6] CORBETT, J. C., DEAN, J., EPSTEIN, M., FIKES, A., FROST, C., FURMAN, J. J., GHEMAWAT, S., GUBAREV, A., HEISER, C., HOCHSCHILD, P., HSIEH, W., KANTHAK, S., KOGAN, E., LI, H., LLOYD, A., MELNIK, S., MWAURA, D., NAGLE, D., QUINLAN, S., RAO, R., ROLIG, L., SAITO, Y., SZYMANIAK, M., TAYLOR, C., WANG, R., AND WOODFORD, D. Spanner: Google's globally-distributed database. In *Proc. OSDI'12, USENIX Conference on Operating Systems Design and Implementation* (2012), USENIX, pp. 251–264.

[7] COUSINEAU, D., DOLIGEZ, D., LAMPORT, L., MERZ, S., RICKETTS, D., AND VANZETTO, H. TLA$^+$ proofs. In *Proc. FM'12, Symposium on Formal Methods* (2012), D. Giannakopoulou and D. Méry, Eds., vol. 7436 of *Lecture Notes in Computer Science*, Springer, pp. 147–154.

[8] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The Google file system. In *Proc. SOSP'03, ACM Symposium on Operating Systems Principles* (2003), ACM, pp. 29–43.

[9] GRAY, C., AND CHERITON, D. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Proceedings of the 12th ACM Ssymposium on Operating Systems Principles* (1989), pp. 202–210.

[10] HERLIHY, M. P., AND WING, J. M. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems 12* (July 1990), 463–492.

[11] HUNT, P., KONAR, M., JUNQUEIRA, F. P., AND REED, B. ZooKeeper: wait-free coordination for internet-scale systems. In *Proc ATC'10, USENIX Annual Technical Conference* (2010), USENIX, pp. 145–158.

[12] JUNQUEIRA, F. P., REED, B. C., AND SERAFINI, M. Zab: High-performance broadcast for primary-backup systems. In *Proc. DSN'11, IEEE/IFIP Int'l Conf. on Dependable Systems & Networks* (2011), IEEE Computer Society, pp. 245–256.

[13] KIRSCH, J., AND AMIR, Y. Paxos for system builders. Tech. Rep. CNDS-2008-2, Johns Hopkins University, 2008.

[14] LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. *Commununications of the ACM 21*, 7 (July 1978), 558–565.

[15] LAMPORT, L. The part-time parliament. *ACM Transactions on Computer Systems 16*, 2 (May 1998), 133–169.

[16] LAMPORT, L. Paxos made simple. *ACM SIGACT News 32*, 4 (Dec. 2001), 18–25.

[17] LAMPORT, L. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[18] LAMPORT, L. Generalized consensus and Paxos. Tech. Rep. MSR-TR-2005-33, Microsoft Research, 2005.

[19] LAMPORT, L. Fast paxos. *Distributed Computing 19*, 2 (2006), 79–103.

[20] LAMPSON, B. W. How to build a highly available system using consensus. In *Distributed Algorithms*, O. Baboaglu and K. Marzullo, Eds. Springer-Verlag, 1996, pp. 1–17.

[21] LAMPSON, B. W. The ABCD's of Paxos. In *Proc. PODC'01, ACM Symposium on Principles of Distributed Computing* (2001), ACM, pp. 13–13.

[22] LISKOV, B., AND COWLING, J. Viewstamped replication revisited. Tech. Rep. MIT-CSAIL-TR-2012-021, MIT, July 2012.

[23] LogCabin source code. `http://github.com/logcabin/logcabin`.

[24] LORCH, J. R., ADYA, A., BOLOSKY, W. J., CHAIKEN, R., DOUCEUR, J. R., AND HOWELL, J. The SMART way to migrate replicated stateful services. In *Proc. EuroSys'06, ACM SIGOPS/EuroSys European Conference on Computer Systems* (2006), ACM, pp. 103–115.

[25] MAO, Y., JUNQUEIRA, F. P., AND MARZULLO, K. Mencius: building efficient replicated state machines for WANs. In *Proc. OSDI'08, USENIX Conference on Operating Systems Design and Implementation* (2008), USENIX, pp. 369–384.

[26] MAZIÈRES, D. Paxos made practical. `http://www.scs.stanford.edu/~dm/home/papers/paxos.pdf`, Jan. 2007.

[27] MORARU, I., ANDERSEN, D. G., AND KAMINSKY, M. There is more consensus in egalitarian parliaments. In *Proc. SOSP'13, ACM Symposium on Operating System Principles* (2013), ACM.

[28] Raft user study. `http://ramcloud.stanford.edu/~ongaro/userstudy/`.

[29] OKI, B. M., AND LISKOV, B. H. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proc. PODC'88, ACM Symposium on Principles of Distributed Computing* (1988), ACM, pp. 8–17.

[30] O'NEIL, P., CHENG, E., GAWLICK, D., AND ONEIL, E. The log-structured merge-tree (LSM-tree). *Acta Informatica 33*, 4 (1996), 351–385.

[31] ONGARO, D. *Consensus: Bridging Theory and Practice*. PhD thesis, Stanford University, 2014 (work in progress). `http://ramcloud.stanford.edu/~ongaro/thesis.pdf`.

[32] ONGARO, D., AND OUSTERHOUT, J. In search of an understandable consensus algorithm. In *Proc ATC'14, USENIX Annual Technical Conference* (2014), USENIX.

[33] OUSTERHOUT, J., AGRAWAL, P., ERICKSON, D., KOZYRAKIS, C., LEVERICH, J., MAZIÈRES, D., MITRA, S., NARAYANAN, A., ONGARO, D., PARULKAR, G., ROSENBLUM, M., RUMBLE, S. M., STRATMANN, E., AND STUTSMAN, R. The case for RAMCloud. *Communications of the ACM 54* (July 2011), 121–130.

[34] Raft consensus algorithm website. `http://raftconsensus.github.io`.

[35] REED, B. Personal communications, May 17, 2013.

[36] ROSENBLUM, M., AND OUSTERHOUT, J. K. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst. 10* (February 1992), 26–52.

[37] SCHNEIDER, F. B. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys 22*, 4 (Dec. 1990), 299–319.

[38] SHVACHKO, K., KUANG, H., RADIA, S., AND CHANSLER, R. The Hadoop distributed file system. In *Proc. MSST'10, Symposium on Mass Storage Systems and Technologies* (2010), IEEE Computer Society, pp. 1–10.

[39] VAN RENESSE, R. Paxos made moderately complex. Tech. rep., Cornell University, 2012.

Chapter 22

Bitcoin: A Peer-to-Peer Electronic Cash System

Authors:
Satoshi Nakamoto

Publication year: 2008

Bitcoin: A Peer-to-Peer Electronic Cash System

Satoshi Nakamoto
satoshin@gmx.com
www.bitcoin.org

Abstract. A purely peer-to-peer version of electronic cash would allow online payments to be sent directly from one party to another without going through a financial institution. Digital signatures provide part of the solution, but the main benefits are lost if a trusted third party is still required to prevent double-spending. We propose a solution to the double-spending problem using a peer-to-peer network. The network timestamps transactions by hashing them into an ongoing chain of hash-based proof-of-work, forming a record that cannot be changed without redoing the proof-of-work. The longest chain not only serves as proof of the sequence of events witnessed, but proof that it came from the largest pool of CPU power. As long as a majority of CPU power is controlled by nodes that are not cooperating to attack the network, they'll generate the longest chain and outpace attackers. The network itself requires minimal structure. Messages are broadcast on a best effort basis, and nodes can leave and rejoin the network at will, accepting the longest proof-of-work chain as proof of what happened while they were gone.

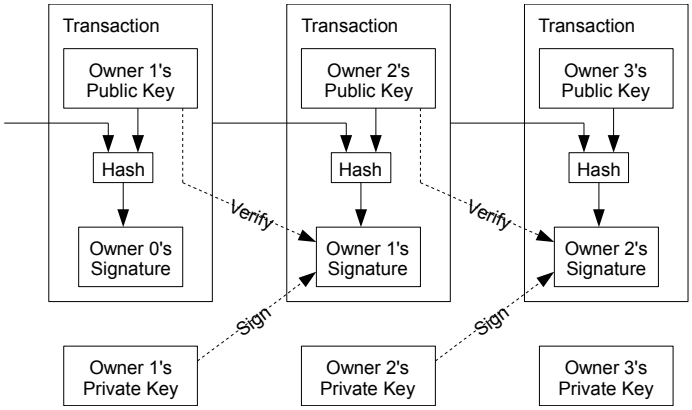
1. Introduction

Commerce on the Internet has come to rely almost exclusively on financial institutions serving as trusted third parties to process electronic payments. While the system works well enough for most transactions, it still suffers from the inherent weaknesses of the trust based model. Completely non-reversible transactions are not really possible, since financial institutions cannot avoid mediating disputes. The cost of mediation increases transaction costs, limiting the minimum practical transaction size and cutting off the possibility for small casual transactions, and there is a broader cost in the loss of ability to make non-reversible payments for non-reversible services. With the possibility of reversal, the need for trust spreads. Merchants must be wary of their customers, hassling them for more information than they would otherwise need. A certain percentage of fraud is accepted as unavoidable. These costs and payment uncertainties can be avoided in person by using physical currency, but no mechanism exists to make payments over a communications channel without a trusted party.

What is needed is an electronic payment system based on cryptographic proof instead of trust, allowing any two willing parties to transact directly with each other without the need for a trusted third party. Transactions that are computationally impractical to reverse would protect sellers from fraud, and routine escrow mechanisms could easily be implemented to protect buyers. In this paper, we propose a solution to the double-spending problem using a peer-to-peer distributed timestamp server to generate computational proof of the chronological order of transactions. The system is secure as long as honest nodes collectively control more CPU power than any cooperating group of attacker nodes.

2. Transactions

We define an electronic coin as a chain of digital signatures. Each owner transfers the coin to the next by digitally signing a hash of the previous transaction and the public key of the next owner and adding these to the end of the coin. A payee can verify the signatures to verify the chain of ownership.

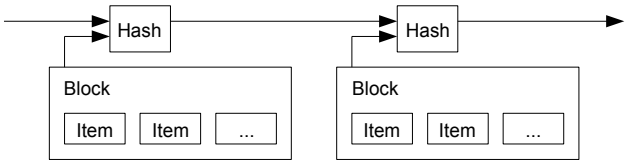


The problem of course is the payee can't verify that one of the owners did not double-spend the coin. A common solution is to introduce a trusted central authority, or mint, that checks every transaction for double spending. After each transaction, the coin must be returned to the mint to issue a new coin, and only coins issued directly from the mint are trusted not to be double-spent. The problem with this solution is that the fate of the entire money system depends on the company running the mint, with every transaction having to go through them, just like a bank.

We need a way for the payee to know that the previous owners did not sign any earlier transactions. For our purposes, the earliest transaction is the one that counts, so we don't care about later attempts to double-spend. The only way to confirm the absence of a transaction is to be aware of all transactions. In the mint based model, the mint was aware of all transactions and decided which arrived first. To accomplish this without a trusted party, transactions must be publicly announced [1], and we need a system for participants to agree on a single history of the order in which they were received. The payee needs proof that at the time of each transaction, the majority of nodes agreed it was the first received.

3. Timestamp Server

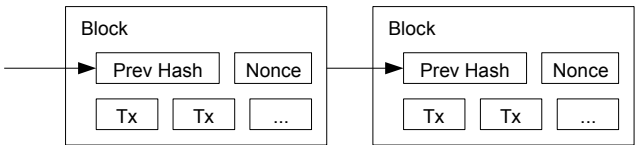
The solution we propose begins with a timestamp server. A timestamp server works by taking a hash of a block of items to be timestamped and widely publishing the hash, such as in a newspaper or Usenet post [2-5]. The timestamp proves that the data must have existed at the time, obviously, in order to get into the hash. Each timestamp includes the previous timestamp in its hash, forming a chain, with each additional timestamp reinforcing the ones before it.



4. Proof-of-Work

To implement a distributed timestamp server on a peer-to-peer basis, we will need to use a proof-of-work system similar to Adam Back's Hashcash [6], rather than newspaper or Usenet posts. The proof-of-work involves scanning for a value that when hashed, such as with SHA-256, the hash begins with a number of zero bits. The average work required is exponential in the number of zero bits required and can be verified by executing a single hash.

For our timestamp network, we implement the proof-of-work by incrementing a nonce in the block until a value is found that gives the block's hash the required zero bits. Once the CPU effort has been expended to make it satisfy the proof-of-work, the block cannot be changed without redoing the work. As later blocks are chained after it, the work to change the block would include redoing all the blocks after it.



The proof-of-work also solves the problem of determining representation in majority decision making. If the majority were based on one-IP-address-one-vote, it could be subverted by anyone able to allocate many IPs. Proof-of-work is essentially one-CPU-one-vote. The majority decision is represented by the longest chain, which has the greatest proof-of-work effort invested in it. If a majority of CPU power is controlled by honest nodes, the honest chain will grow the fastest and outpace any competing chains. To modify a past block, an attacker would have to redo the proof-of-work of the block and all blocks after it and then catch up with and surpass the work of the honest nodes. We will show later that the probability of a slower attacker catching up diminishes exponentially as subsequent blocks are added.

To compensate for increasing hardware speed and varying interest in running nodes over time, the proof-of-work difficulty is determined by a moving average targeting an average number of blocks per hour. If they're generated too fast, the difficulty increases.

5. Network

The steps to run the network are as follows:

1) New transactions are broadcast to all nodes.
2) Each node collects new transactions into a block.
3) Each node works on finding a difficult proof-of-work for its block.
4) When a node finds a proof-of-work, it broadcasts the block to all nodes.
5) Nodes accept the block only if all transactions in it are valid and not already spent.
6) Nodes express their acceptance of the block by working on creating the next block in the chain, using the hash of the accepted block as the previous hash.

Nodes always consider the longest chain to be the correct one and will keep working on extending it. If two nodes broadcast different versions of the next block simultaneously, some nodes may receive one or the other first. In that case, they work on the first one they received, but save the other branch in case it becomes longer. The tie will be broken when the next proof-of-work is found and one branch becomes longer; the nodes that were working on the other branch will then switch to the longer one.

New transaction broadcasts do not necessarily need to reach all nodes. As long as they reach many nodes, they will get into a block before long. Block broadcasts are also tolerant of dropped messages. If a node does not receive a block, it will request it when it receives the next block and realizes it missed one.

6. Incentive

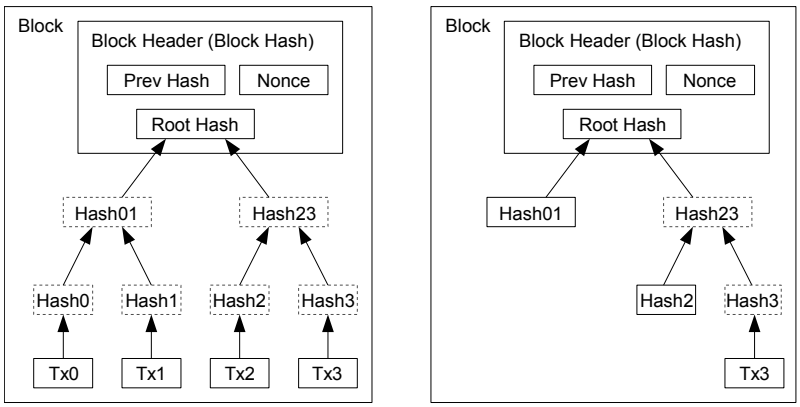
By convention, the first transaction in a block is a special transaction that starts a new coin owned by the creator of the block. This adds an incentive for nodes to support the network, and provides a way to initially distribute coins into circulation, since there is no central authority to issue them. The steady addition of a constant of amount of new coins is analogous to gold miners expending resources to add gold to circulation. In our case, it is CPU time and electricity that is expended.

The incentive can also be funded with transaction fees. If the output value of a transaction is less than its input value, the difference is a transaction fee that is added to the incentive value of the block containing the transaction. Once a predetermined number of coins have entered circulation, the incentive can transition entirely to transaction fees and be completely inflation free.

The incentive may help encourage nodes to stay honest. If a greedy attacker is able to assemble more CPU power than all the honest nodes, he would have to choose between using it to defraud people by stealing back his payments, or using it to generate new coins. He ought to find it more profitable to play by the rules, such rules that favour him with more new coins than everyone else combined, than to undermine the system and the validity of his own wealth.

7. Reclaiming Disk Space

Once the latest transaction in a coin is buried under enough blocks, the spent transactions before it can be discarded to save disk space. To facilitate this without breaking the block's hash, transactions are hashed in a Merkle Tree [7][2][5], with only the root included in the block's hash. Old blocks can then be compacted by stubbing off branches of the tree. The interior hashes do not need to be stored.



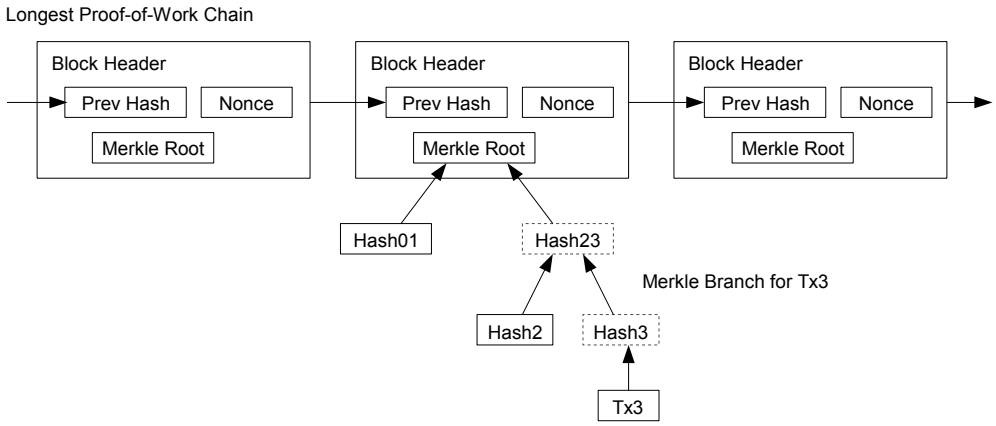
Transactions Hashed in a Merkle Tree

After Pruning Tx0-2 from the Block

A block header with no transactions would be about 80 bytes. If we suppose blocks are generated every 10 minutes, 80 bytes * 6 * 24 * 365 = 4.2MB per year. With computer systems typically selling with 2GB of RAM as of 2008, and Moore's Law predicting current growth of 1.2GB per year, storage should not be a problem even if the block headers must be kept in memory.

8. Simplified Payment Verification

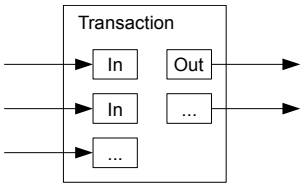
It is possible to verify payments without running a full network node. A user only needs to keep a copy of the block headers of the longest proof-of-work chain, which he can get by querying network nodes until he's convinced he has the longest chain, and obtain the Merkle branch linking the transaction to the block it's timestamped in. He can't check the transaction for himself, but by linking it to a place in the chain, he can see that a network node has accepted it, and blocks added after it further confirm the network has accepted it.



As such, the verification is reliable as long as honest nodes control the network, but is more vulnerable if the network is overpowered by an attacker. While network nodes can verify transactions for themselves, the simplified method can be fooled by an attacker's fabricated transactions for as long as the attacker can continue to overpower the network. One strategy to protect against this would be to accept alerts from network nodes when they detect an invalid block, prompting the user's software to download the full block and alerted transactions to confirm the inconsistency. Businesses that receive frequent payments will probably still want to run their own nodes for more independent security and quicker verification.

9. Combining and Splitting Value

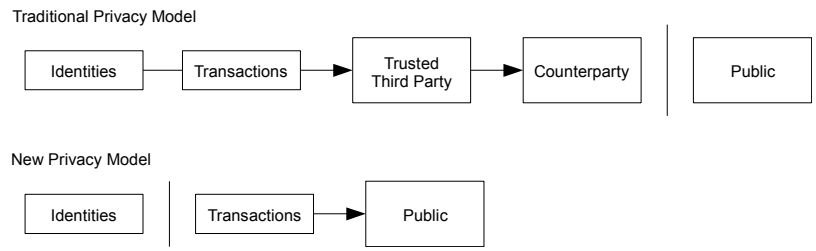
Although it would be possible to handle coins individually, it would be unwieldy to make a separate transaction for every cent in a transfer. To allow value to be split and combined, transactions contain multiple inputs and outputs. Normally there will be either a single input from a larger previous transaction or multiple inputs combining smaller amounts, and at most two outputs: one for the payment, and one returning the change, if any, back to the sender.



It should be noted that fan-out, where a transaction depends on several transactions, and those transactions depend on many more, is not a problem here. There is never the need to extract a complete standalone copy of a transaction's history.

10. Privacy

The traditional banking model achieves a level of privacy by limiting access to information to the parties involved and the trusted third party. The necessity to announce all transactions publicly precludes this method, but privacy can still be maintained by breaking the flow of information in another place: by keeping public keys anonymous. The public can see that someone is sending an amount to someone else, but without information linking the transaction to anyone. This is similar to the level of information released by stock exchanges, where the time and size of individual trades, the "tape", is made public, but without telling who the parties were.



As an additional firewall, a new key pair should be used for each transaction to keep them from being linked to a common owner. Some linking is still unavoidable with multi-input transactions, which necessarily reveal that their inputs were owned by the same owner. The risk is that if the owner of a key is revealed, linking could reveal other transactions that belonged to the same owner.

11. Calculations

We consider the scenario of an attacker trying to generate an alternate chain faster than the honest chain. Even if this is accomplished, it does not throw the system open to arbitrary changes, such as creating value out of thin air or taking money that never belonged to the attacker. Nodes are not going to accept an invalid transaction as payment, and honest nodes will never accept a block containing them. An attacker can only try to change one of his own transactions to take back money he recently spent.

The race between the honest chain and an attacker chain can be characterized as a Binomial Random Walk. The success event is the honest chain being extended by one block, increasing its lead by +1, and the failure event is the attacker's chain being extended by one block, reducing the gap by -1.

The probability of an attacker catching up from a given deficit is analogous to a Gambler's Ruin problem. Suppose a gambler with unlimited credit starts at a deficit and plays potentially an infinite number of trials to try to reach breakeven. We can calculate the probability he ever reaches breakeven, or that an attacker ever catches up with the honest chain, as follows [8]:

p = probability an honest node finds the next block
q = probability the attacker finds the next block
q_z = probability the attacker will ever catch up from z blocks behind

$$q_z = \begin{cases} 1 & \text{if } p \le q \\ (q/p)^z & \text{if } p > q \end{cases}$$

Given our assumption that $p > q$, the probability drops exponentially as the number of blocks the attacker has to catch up with increases. With the odds against him, if he doesn't make a lucky lunge forward early on, his chances become vanishingly small as he falls further behind.

We now consider how long the recipient of a new transaction needs to wait before being sufficiently certain the sender can't change the transaction. We assume the sender is an attacker who wants to make the recipient believe he paid him for a while, then switch it to pay back to himself after some time has passed. The receiver will be alerted when that happens, but the sender hopes it will be too late.

The receiver generates a new key pair and gives the public key to the sender shortly before signing. This prevents the sender from preparing a chain of blocks ahead of time by working on it continuously until he is lucky enough to get far enough ahead, then executing the transaction at that moment. Once the transaction is sent, the dishonest sender starts working in secret on a parallel chain containing an alternate version of his transaction.

The recipient waits until the transaction has been added to a block and z blocks have been linked after it. He doesn't know the exact amount of progress the attacker has made, but assuming the honest blocks took the average expected time per block, the attacker's potential progress will be a Poisson distribution with expected value:

$$\lambda = z \frac{q}{p}$$

To get the probability the attacker could still catch up now, we multiply the Poisson density for each amount of progress he could have made by the probability he could catch up from that point:

$$\sum_{k=0}^{\infty} \frac{\lambda^k e^{-\lambda}}{k!} \cdot \begin{Bmatrix} (q/p)^{(z-k)} & \text{if } k \le z \\ 1 & \text{if } k > z \end{Bmatrix}$$

Rearranging to avoid summing the infinite tail of the distribution...

$$1 - \sum_{k=0}^{z} \frac{\lambda^k e^{-\lambda}}{k!} \left(1 - (q/p)^{(z-k)}\right)$$

Converting to C code...

```
#include <math.h>
double AttackerSuccessProbability(double q, int z)
{
    double p = 1.0 - q;
    double lambda = z * (q / p);
    double sum = 1.0;
    int i, k;
    for (k = 0; k <= z; k++)
    {
        double poisson = exp(-lambda);
        for (i = 1; i <= k; i++)
            poisson *= lambda / i;
        sum -= poisson * (1 - pow(q / p, z - k));
    }
    return sum;
}
```

Running some results, we can see the probability drop off exponentially with z.

```
q=0.1
z=0    P=1.0000000
z=1    P=0.2045873
z=2    P=0.0509779
z=3    P=0.0131722
z=4    P=0.0034552
z=5    P=0.0009137
z=6    P=0.0002428
z=7    P=0.0000647
z=8    P=0.0000173
z=9    P=0.0000046
z=10   P=0.0000012

q=0.3
z=0    P=1.0000000
z=5    P=0.1773523
z=10   P=0.0416605
z=15   P=0.0101008
z=20   P=0.0024804
z=25   P=0.0006132
z=30   P=0.0001522
z=35   P=0.0000379
z=40   P=0.0000095
z=45   P=0.0000024
z=50   P=0.0000006
```

Solving for P less than 0.1%...

```
P < 0.001
q=0.10   z=5
q=0.15   z=8
q=0.20   z=11
q=0.25   z=15
q=0.30   z=24
q=0.35   z=41
q=0.40   z=89
q=0.45   z=340
```

12. Conclusion

We have proposed a system for electronic transactions without relying on trust. We started with the usual framework of coins made from digital signatures, which provides strong control of ownership, but is incomplete without a way to prevent double-spending. To solve this, we proposed a peer-to-peer network using proof-of-work to record a public history of transactions that quickly becomes computationally impractical for an attacker to change if honest nodes control a majority of CPU power. The network is robust in its unstructured simplicity. Nodes work all at once with little coordination. They do not need to be identified, since messages are not routed to any particular place and only need to be delivered on a best effort basis. Nodes can leave and rejoin the network at will, accepting the proof-of-work chain as proof of what happened while they were gone. They vote with their CPU power, expressing their acceptance of valid blocks by working on extending them and rejecting invalid blocks by refusing to work on them. Any needed rules and incentives can be enforced with this consensus mechanism.

References

[1] W. Dai, "b-money," http://www.weidai.com/bmoney.txt, 1998.

[2] H. Massias, X.S. Avila, and J.-J. Quisquater, "Design of a secure timestamping service with minimal trust requirements," In *20th Symposium on Information Theory in the Benelux*, May 1999.

[3] S. Haber, W.S. Stornetta, "How to time-stamp a digital document," In *Journal of Cryptology*, vol 3, no 2, pages 99-111, 1991.

[4] D. Bayer, S. Haber, W.S. Stornetta, "Improving the efficiency and reliability of digital time-stamping," In *Sequences II: Methods in Communication, Security and Computer Science*, pages 329-334, 1993.

[5] S. Haber, W.S. Stornetta, "Secure names for bit-strings," In *Proceedings of the 4th ACM Conference on Computer and Communications Security*, pages 28-35, April 1997.

[6] A. Back, "Hashcash - a denial of service counter-measure," http://www.hashcash.org/papers/hashcash.pdf, 2002.

[7] R.C. Merkle, "Protocols for public key cryptosystems," In *Proc. 1980 Symposium on Security and Privacy*, IEEE Computer Society, pages 122-133, April 1980.

[8] W. Feller, "An introduction to probability theory and its applications," 1957.

Chapter 23

Ethereum: a secure decentralized generalized transaction ledger

Authors:
Gavin Wood

Publication year: 2015

ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER

BYZANTIUM VERSION e94ebda - 2018-06-05

DR. GAVIN WOOD
FOUNDER, ETHEREUM & PARITY
GAVIN@PARITY.IO

Abstract. The blockchain paradigm when coupled with cryptographically-secured transactions has demonstrated its utility through a number of projects, with Bitcoin being one of the most notable ones. Each such project can be seen as a simple application on a decentralised, but singleton, compute resource. We can call this paradigm a transactional singleton machine with shared-state.

Ethereum implements this paradigm in a generalised manner. Furthermore it provides a plurality of such resources, each with a distinct state and operating code but able to interact through a message-passing framework with others. We discuss its design, implementation issues, the opportunities it provides and the future hurdles we envisage.

1. Introduction

With ubiquitous internet connections in most places of the world, global information transmission has become incredibly cheap. Technology-rooted movements like Bitcoin have demonstrated through the power of the default, consensus mechanisms, and voluntary respect of the social contract, that it is possible to use the internet to make a decentralised value-transfer system that can be shared across the world and virtually free to use. This system can be said to be a very specialised version of a cryptographically secure, transaction-based state machine. Follow-up systems such as Namecoin adapted this original "currency application" of the technology into other applications albeit rather simplistic ones.

Ethereum is a project which attempts to build the generalised technology; technology on which all transaction-based state machine concepts may be built. Moreover it aims to provide to the end-developer a tightly integrated end-to-end system for building software on a hitherto unexplored compute paradigm in the mainstream: a trustful object messaging compute framework.

1.1. Driving Factors.

There are many goals of this project; one key goal is to facilitate transactions between consenting individuals who would otherwise have no means to trust one another. This may be due to geographical separation, interfacing difficulty, or perhaps the incompatibility, incompetence, unwillingness, expense, uncertainty, inconvenience, or corruption of existing legal systems. By specifying a state-change system through a rich and unambiguous language, and furthermore architecting a system such that we can reasonably expect that an agreement will be thus enforced autonomously, we can provide a means to this end.

Dealings in this proposed system would have several attributes not often found in the real world. The incorruptibility of judgement, often difficult to find, comes naturally from a disinterested algorithmic interpreter. Transparency, or being able to see exactly how a state or judgement came about through the transaction log and rules or instructional codes, never happens perfectly in human-based systems since natural language is necessarily vague, information is often lacking, and plain old prejudices are difficult to shake.

Overall, we wish to provide a system such that users can be guaranteed that no matter with which other individuals, systems or organisations they interact, they can do so with absolute confidence in the possible outcomes and how those outcomes might come about.

1.2. Previous Work.

Buterin [2013a] first proposed the kernel of this work in late November, 2013. Though now evolved in many ways, the key functionality of a blockchain with a Turing-complete language and an effectively unlimited inter-transaction storage capability remains unchanged.

Dwork and Naor [1992] provided the first work into the usage of a cryptographic proof of computational expenditure ("proof-of-work") as a means of transmitting a value signal over the Internet. The value-signal was utilised here as a spam deterrence mechanism rather than any kind of currency, but critically demonstrated the potential for a basic data channel to carry a *strong economic signal*, allowing a receiver to make a physical assertion without having to rely upon *trust*. Back [2002] later produced a system in a similar vein.

The first example of utilising the proof-of-work as a strong economic signal to secure a currency was by Vishnumurthy et al. [2003]. In this instance, the token was used to keep peer-to-peer file trading in check, providing "consumers" with the ability to make micro-payments to "suppliers" for their services. The security model afforded by the proof-of-work was augmented with digital signatures and a ledger in order to ensure that the historical record couldn't be corrupted and that malicious actors could not spoof payment or unjustly complain about service delivery. Five years later, Nakamoto [2008] introduced another such proof-of-work-secured value token, somewhat wider in scope. The fruits of this project, Bitcoin, became the first widely adopted global decentralised transaction ledger.

Other projects built on Bitcoin's success; the alt-coins introduced numerous other currencies through alteration to the protocol. Some of the best known are Litecoin and Primecoin, discussed by Sprankel [2013]. Other projects sought to take the core value content mechanism of the protocol and repurpose it; Aron [2012] discusses, for example,

the Namecoin project which aims to provide a decentralised name-resolution system.

Other projects still aim to build upon the Bitcoin network itself, leveraging the large amount of value placed in the system and the vast amount of computation that goes into the consensus mechanism. The Mastercoin project, first proposed by Willett [2013], aims to build a richer protocol involving many additional high-level features on top of the Bitcoin protocol through utilisation of a number of auxiliary parts to the core protocol. The Coloured Coins project, proposed by Rosenfeld et al. [2012], takes a similar but more simplified strategy, embellishing the rules of a transaction in order to break the fungibility of Bitcoin's base currency and allow the creation and tracking of tokens through a special "chroma-wallet"-protocol-aware piece of software.

Additional work has been done in the area with discarding the decentralisation foundation; Ripple, discussed by Boutellier and Heinzen [2014], has sought to create a "federated" system for currency exchange, effectively creating a new financial clearing system. It has demonstrated that high efficiency gains can be made if the decentralisation premise is discarded.

Early work on smart contracts has been done by Szabo [1997] and Miller [1997]. Around the 1990s it became clear that algorithmic enforcement of agreements could become a significant force in human cooperation. Though no specific system was proposed to implement such a system, it was proposed that the future of law would be heavily affected by such systems. In this light, Ethereum may be seen as a general implementation of such a *crypto-law* system.

For a list of terms used in this paper, refer to Appendix A.

2. The Blockchain Paradigm

Ethereum, taken as a whole, can be viewed as a transaction-based state machine: we begin with a genesis state and incrementally execute transactions to morph it into some final state. It is this final state which we accept as the canonical "version" of the world of Ethereum. The state can include such information as account balances, reputations, trust arrangements, data pertaining to information of the physical world; in short, anything that can currently be represented by a computer is admissible. Transactions thus represent a valid arc between two states; the 'valid' part is important—there exist far more invalid state changes than valid state changes. Invalid state changes might, e.g., be things such as reducing an account balance without an equal and opposite increase elsewhere. A valid state transition is one which comes about through a transaction. Formally:

$$\boldsymbol{\sigma}_{t+1} \equiv \Upsilon(\boldsymbol{\sigma}_t, T) \tag{1}$$

where Υ is the Ethereum state transition function. In Ethereum, Υ, together with $\boldsymbol{\sigma}$ are considerably more powerful than any existing comparable system; Υ allows components to carry out arbitrary computation, while $\boldsymbol{\sigma}$ allows components to store arbitrary state between transactions.

Transactions are collated into blocks; blocks are chained together using a cryptographic hash as a means of reference. Blocks function as a journal, recording a series of transactions together with the previous block and an identifier for the final state (though do not store the final state itself—that would be far too big). They also punctuate the transaction series with incentives for nodes to *mine*. This incentivisation takes place as a state-transition function, adding value to a nominated account.

Mining is the process of dedicating effort (working) to bolster one series of transactions (a block) over any other potential competitor block. It is achieved thanks to a cryptographically secure proof. This scheme is known as a proof-of-work and is discussed in detail in section 11.5.

Formally, we expand to:

$$\boldsymbol{\sigma}_{t+1} \equiv \Pi(\boldsymbol{\sigma}_t, B) \tag{2}$$

$$B \equiv (..., (T_0, T_1, ...), ...) \tag{3}$$

$$\Pi(\boldsymbol{\sigma}, B) \equiv \Omega(B, \Upsilon(\Upsilon(\boldsymbol{\sigma}, T_0), T_1)...) \tag{4}$$

Where Ω is the block-finalisation state transition function (a function that rewards a nominated party); B is this block, which includes a series of transactions amongst some other components; and Π is the block-level state-transition function.

This is the basis of the blockchain paradigm, a model that forms the backbone of not only Ethereum, but all decentralised consensus-based transaction systems to date.

2.1. Value.

In order to incentivise computation within the network, there needs to be an agreed method for transmitting value. To address this issue, Ethereum has an intrinsic currency, Ether, known also as ETH and sometimes referred to by the Old English Ð. The smallest subdenomination of Ether, and thus the one in which all integer values of the currency are counted, is the Wei. One Ether is defined as being 10^{18} Wei. There exist other subdenominations of Ether:

Multiplier	Name
10^0	Wei
10^{12}	Szabo
10^{15}	Finney
10^{18}	Ether

Throughout the present work, any reference to value, in the context of Ether, currency, a balance or a payment, should be assumed to be counted in Wei.

2.2. Which History?

Since the system is decentralised and all parties have an opportunity to create a new block on some older pre-existing block, the resultant structure is necessarily a tree of blocks. In order to form a consensus as to which path, from root (the genesis block) to leaf (the block containing the most recent transactions) through this tree structure, known as the blockchain, there must be an agreed-upon scheme. If there is ever a disagreement between nodes as to which root-to-leaf path down the block tree is the 'best' blockchain, then a *fork* occurs.

This would mean that past a given point in time (block), multiple states of the system may coexist: some nodes believing one block to contain the canonical transactions, other nodes believing some other block to be canonical, potentially containing radically different or incompatible transactions. This is to be avoided at all costs as the uncertainty that would ensue would likely kill all confidence in the entire system.

The scheme we use in order to generate consensus is a simplified version of the GHOST protocol introduced by Sompolinsky and Zohar [2013]. This process is described in detail in section 10.

Sometimes, a path follows a new protocol from a particular height. This document describes one version of the protocol. In order to follow back the history of a path, one must reference multiple versions of this document.

3. Conventions

We use a number of typographical conventions for the formal notation, some of which are quite particular to the present work:

The two sets of highly structured, 'top-level', state values, are denoted with bold lowercase Greek letters. They fall into those of world-state, which are denoted $\boldsymbol{\sigma}$ (or a variant thereupon) and those of machine-state, $\boldsymbol{\mu}$.

Functions operating on highly structured values are denoted with an upper-case Greek letter, e.g. Υ, the Ethereum state transition function.

For most functions, an uppercase letter is used, e.g. C, the general cost function. These may be subscripted to denote specialised variants, e.g. C_{SSTORE}, the cost function for the SSTORE operation. For specialised and possibly externally defined functions, we may format as typewriter text, e.g. the Keccak-256 hash function (as per the winning entry to the SHA-3 contest by Bertoni et al. [2017], rather than later releases), is denoted `KEC` (and generally referred to as plain Keccak). Also `KEC512` is referring to the Keccak 512 hash function.

Tuples are typically denoted with an upper-case letter, e.g. T, is used to denote an Ethereum transaction. This symbol may, if accordingly defined, be subscripted to refer to an individual component, e.g. T_{n}, denotes the nonce of said transaction. The form of the subscript is used to denote its type; e.g. uppercase subscripts refer to tuples with subscriptable components.

Scalars and fixed-size byte sequences (or, synonymously, arrays) are denoted with a normal lower-case letter, e.g. n is used in the document to denote a transaction nonce. Those with a particularly special meaning may be Greek, e.g. δ, the number of items required on the stack for a given operation.

Arbitrary-length sequences are typically denoted as a bold lower-case letter, e.g. $\mathbf{o}$ is used to denote the byte sequence given as the output data of a message call. For particularly important values, a bold uppercase letter may be used.

Throughout, we assume scalars are non-negative integers and thus belong to the set $\mathbb{N}$. The set of all byte sequences is $\mathbb{B}$, formally defined in Appendix B. If such a set of sequences is restricted to those of a particular length, it is denoted with a subscript, thus the set of all byte sequences of length 32 is named $\mathbb{B}_{32}$ and the set of all non-negative integers smaller than 2^{256} is named $\mathbb{N}_{256}$. This is formally defined in section 4.3.

Square brackets are used to index into and reference individual components or subsequences of sequences, e.g. $\boldsymbol{\mu}_{\mathbf{s}}[0]$ denotes the first item on the machine's stack. For subsequences, ellipses are used to specify the intended range, to include elements at both limits, e.g. $\boldsymbol{\mu}_{\mathbf{m}}[0..31]$ denotes the first 32 items of the machine's memory.

In the case of the global state $\boldsymbol{\sigma}$, which is a sequence of accounts, themselves tuples, the square brackets are used to reference an individual account.

When considering variants of existing values, we follow the rule that within a given scope for definition, if we assume that the unmodified 'input' value be denoted by the placeholder $\square$ then the modified and utilisable value is denoted as $\square'$, and intermediate values would be $\square^*$, $\square^{**}$ &c. On very particular occasions, in order to maximise readability and only if unambiguous in meaning, we may use alpha-numeric subscripts to denote intermediate values, especially those of particular note.

When considering the use of existing functions, given a function f, the function f^* denotes a similar, element-wise version of the function mapping instead between sequences. It is formally defined in section 4.3.

We define a number of useful functions throughout. One of the more common is ℓ, which evaluates to the last item in the given sequence:

$$\ell(\mathbf{x}) \equiv \mathbf{x}[\|\mathbf{x}\| - 1] \tag{5}$$

4. Blocks, State and Transactions

Having introduced the basic concepts behind Ethereum, we will discuss the meaning of a transaction, a block and the state in more detail.

4.1. World State.

The world state (*state*), is a mapping between addresses (160-bit identifiers) and account states (a data structure serialised as RLP, see Appendix B). Though not stored on the blockchain, it is assumed that the implementation will maintain this mapping in a modified Merkle Patricia tree (*trie*, see Appendix D). The trie requires a simple database backend that maintains a mapping of bytearrays to bytearrays; we name this underlying database the state database. This has a number of benefits; firstly the root node of this structure is cryptographically dependent on all internal data and as such its hash can be used as a secure identity for the entire system state. Secondly, being an immutable data structure, it allows any previous state (whose root hash is known) to be recalled by simply altering the root hash accordingly. Since we store all such root hashes in the blockchain, we are able to trivially revert to old states.

The account state, $\boldsymbol{\sigma}[a]$, comprises the following four fields:

- **nonce:** A scalar value equal to the number of transactions sent from this address or, in the case of accounts with associated code, the number of contract-creations made by this account. For account of address a in state $\boldsymbol{\sigma}$, this would be formally denoted $\boldsymbol{\sigma}[a]_{\text{n}}$.
- **balance:** A scalar value equal to the number of Wei owned by this address. Formally denoted $\boldsymbol{\sigma}[a]_{\text{b}}$.
- **storageRoot:** A 256-bit hash of the root node of a Merkle Patricia tree that encodes the storage contents of the account (a mapping between 256-bit integer values), encoded into the trie as a mapping from the Keccak 256-bit hash of the 256-bit integer keys to the RLP-encoded 256-bit integer values. The hash is formally denoted $\boldsymbol{\sigma}[a]_{\text{s}}$.
- **codeHash:** The hash of the EVM code of this account—this is the code that gets executed should this address receive a message call; it is immutable and thus, unlike all other fields, cannot be changed after construction. All such code fragments are contained in the state database under their corresponding hashes for later retrieval. This hash is

formally denoted $\boldsymbol{\sigma}[a]_{\mathrm{c}}$, and thus the code may be denoted as $\mathbf{b}$, given that $\mathtt{KEC}(\mathbf{b}) = \boldsymbol{\sigma}[a]_{\mathrm{c}}$.

Since we typically wish to refer not to the trie's root hash but to the underlying set of key/value pairs stored within, we define a convenient equivalence:

$$\mathtt{TRIE}\big(L_I^*(\boldsymbol{\sigma}[a]_{\mathbf{s}})\big) \equiv \boldsymbol{\sigma}[a]_{\mathrm{s}} \tag{6}$$

The collapse function for the set of key/value pairs in the trie, L_I^*, is defined as the element-wise transformation of the base function L_I, given as:

$$L_I\big((k, v)\big) \equiv \big(\mathtt{KEC}(k), \mathtt{RLP}(v)\big) \tag{7}$$

where:

$$k \in \mathbb{B}_{32} \quad \wedge \quad v \in \mathbb{N} \tag{8}$$

It shall be understood that $\boldsymbol{\sigma}[a]_{\mathbf{s}}$ is not a 'physical' member of the account and does not contribute to its later serialisation.

If the **codeHash** field is the Keccak-256 hash of the empty string, i.e. $\boldsymbol{\sigma}[a]_{\mathrm{c}} = \mathtt{KEC}\big(()\big)$, then the node represents a simple account, sometimes referred to as a "non-contract" account.

Thus we may define a world-state collapse function L_S:

$$L_S(\boldsymbol{\sigma}) \equiv \{p(a) : \boldsymbol{\sigma}[a] \neq \varnothing\} \tag{9}$$

where

$$p(a) \equiv \big(\mathtt{KEC}(a), \mathtt{RLP}\big((\boldsymbol{\sigma}[a]_{\mathrm{n}}, \boldsymbol{\sigma}[a]_{\mathrm{b}}, \boldsymbol{\sigma}[a]_{\mathrm{s}}, \boldsymbol{\sigma}[a]_{\mathrm{c}})\big)\big) \tag{10}$$

This function, L_S, is used alongside the trie function to provide a short identity (hash) of the world state. We assume:

$$\forall a : \boldsymbol{\sigma}[a] = \varnothing \ \vee \ (a \in \mathbb{B}_{20} \ \wedge \ v(\boldsymbol{\sigma}[a])) \tag{11}$$

where v is the account validity function:

$$v(x) \equiv x_{\mathrm{n}} \in \mathbb{N}_{256} \wedge x_{\mathrm{b}} \in \mathbb{N}_{256} \wedge x_{\mathrm{s}} \in \mathbb{B}_{32} \wedge x_{\mathrm{c}} \in \mathbb{B}_{32} \tag{12}$$

An account is *empty* when it has no code, zero nonce and zero balance:

$$\mathtt{EMPTY}(\boldsymbol{\sigma}, a) \quad \equiv \quad \boldsymbol{\sigma}[a]_{\mathrm{c}} = \mathtt{KEC}\big(()\big) \wedge \boldsymbol{\sigma}[a]_{\mathrm{n}} = 0 \wedge \boldsymbol{\sigma}[a]_{\mathrm{b}} = 0 \tag{13}$$

Even callable precompiled contracts can have an empty account state. This is because their account states do not usually contain the code describing its behavior.

An account is *dead* when its account state is non-existent or empty:

$$\mathtt{DEAD}(\boldsymbol{\sigma}, a) \quad \equiv \quad \boldsymbol{\sigma}[a] = \varnothing \vee \mathtt{EMPTY}(\boldsymbol{\sigma}, a) \tag{14}$$

4.2. **The Transaction.** A transaction (formally, T) is a single cryptographically-signed instruction constructed by an actor externally to the scope of Ethereum. While it is assumed that the ultimate external actor will be human in nature, software tools will be used in its construction and dissemination[1]. There are two types of transactions: those which result in message calls and those which result in the creation of new accounts with associated code (known informally as 'contract creation'). Both types specify a number of common fields:

nonce: A scalar value equal to the number of transactions sent by the sender; formally T_{n}.

gasPrice: A scalar value equal to the number of Wei to be paid per unit of *gas* for all computation costs incurred as a result of the execution of this transaction; formally T_{p}.

gasLimit: A scalar value equal to the maximum amount of gas that should be used in executing this transaction. This is paid up-front, before any computation is done and may not be increased later; formally T_{g}.

to: The 160-bit address of the message call's recipient or, for a contract creation transaction, $\varnothing$, used here to denote the only member of $\mathbb{B}_0$; formally T_{t}.

value: A scalar value equal to the number of Wei to be transferred to the message call's recipient or, in the case of contract creation, as an endowment to the newly created account; formally T_{v}.

v, r, s: Values corresponding to the signature of the transaction and used to determine the sender of the transaction; formally T_{w}, T_{r} and T_{s}. This is expanded in Appendix F.

Additionally, a contract creation transaction contains:

init: An unlimited size byte array specifying the EVM-code for the account initialisation procedure, formally $T_{\mathbf{i}}$.

init is an EVM-code fragment; it returns the **body**, a second fragment of code that executes each time the account receives a message call (either through a transaction or due to the internal execution of code). **init** is executed only once at account creation and gets discarded immediately thereafter.

In contrast, a message call transaction contains:

data: An unlimited size byte array specifying the input data of the message call, formally $T_{\mathbf{d}}$.

Appendix F specifies the function, S, which maps transactions to the sender, and happens through the ECDSA of the SECP-256k1 curve, using the hash of the transaction (excepting the latter three signature fields) as the datum to sign. For the present we simply assert that the sender of a given transaction T can be represented with $S(T)$.

$$L_T(T) \equiv \begin{cases} (T_{\mathrm{n}}, T_{\mathrm{p}}, T_{\mathrm{g}}, T_{\mathrm{t}}, T_{\mathrm{v}}, T_{\mathbf{i}}, T_{\mathrm{w}}, T_{\mathrm{r}}, T_{\mathrm{s}}) & \text{if } T_{\mathrm{t}} = \varnothing \\ (T_{\mathrm{n}}, T_{\mathrm{p}}, T_{\mathrm{g}}, T_{\mathrm{t}}, T_{\mathrm{v}}, T_{\mathbf{d}}, T_{\mathrm{w}}, T_{\mathrm{r}}, T_{\mathrm{s}}) & \text{otherwise} \end{cases} \tag{15}$$

Here, we assume all components are interpreted by the RLP as integer values, with the exception of the arbitrary length byte arrays $T_{\mathbf{i}}$ and $T_{\mathbf{d}}$.

$$\begin{aligned} &T_{\mathrm{n}} \in \mathbb{N}_{256} \quad \wedge \quad T_{\mathrm{v}} \in \mathbb{N}_{256} \quad \wedge \quad T_{\mathrm{p}} \in \mathbb{N}_{256} \quad \wedge \\ &T_{\mathrm{g}} \in \mathbb{N}_{256} \quad \wedge \quad T_{\mathrm{w}} \in \mathbb{N}_{5} \quad \wedge \quad T_{\mathrm{r}} \in \mathbb{N}_{256} \quad \wedge \\ &T_{\mathrm{s}} \in \mathbb{N}_{256} \quad \wedge \quad T_{\mathbf{d}} \in \mathbb{B} \quad \wedge \quad T_{\mathbf{i}} \in \mathbb{B} \end{aligned} \tag{16}$$

where

$$\mathbb{N}_{\mathrm{n}} = \{P : P \in \mathbb{N} \wedge P < 2^n\} \tag{17}$$

The address hash $T_{\mathbf{t}}$ is slightly different: it is either a 20-byte address hash or, in the case of being a contract-creation transaction (and thus formally equal to $\varnothing$), it is

[1]Notably, such 'tools' could ultimately become so causally removed from their human-based initiation—or humans may become so causally-neutral—that there could be a point at which they rightly be considered autonomous agents. e.g. contracts may offer bounties to humans for being sent transactions to initiate their execution.

the RLP empty byte sequence and thus the member of $\mathbb{B}_0$:

$$T_{\mathbf{t}} \in \begin{cases} \mathbb{B}_{20} & \text{if} \quad T_{\mathrm{t}} \neq \varnothing \\ \mathbb{B}_0 & \text{otherwise} \end{cases} \tag{18}$$

4.3. **The Block.** The block in Ethereum is the collection of relevant pieces of information (known as the block *header*), H, together with information corresponding to the comprised transactions, $\mathbf{T}$, and a set of other block headers $\mathbf{U}$ that are known to have a parent equal to the present block's parent's parent (such blocks are known as *ommers*[2]). The block header contains several pieces of information:

- **parentHash:** The Keccak 256-bit hash of the parent block's header, in its entirety; formally H_{p}.
- **ommersHash:** The Keccak 256-bit hash of the ommers list portion of this block; formally H_{o}.
- **beneficiary:** The 160-bit address to which all fees collected from the successful mining of this block be transferred; formally H_{c}.
- **stateRoot:** The Keccak 256-bit hash of the root node of the state trie, after all transactions are executed and finalisations applied; formally H_{r}.
- **transactionsRoot:** The Keccak 256-bit hash of the root node of the trie structure populated with each transaction in the transactions list portion of the block; formally H_{t}.
- **receiptsRoot:** The Keccak 256-bit hash of the root node of the trie structure populated with the receipts of each transaction in the transactions list portion of the block; formally H_{e}.
- **logsBloom:** The Bloom filter composed from indexable information (logger address and log topics) contained in each log entry from the receipt of each transaction in the transactions list; formally H_{b}.
- **difficulty:** A scalar value corresponding to the difficulty level of this block. This can be calculated from the previous block's difficulty level and the timestamp; formally H_{d}.
- **number:** A scalar value equal to the number of ancestor blocks. The genesis block has a number of zero; formally H_{i}.
- **gasLimit:** A scalar value equal to the current limit of gas expenditure per block; formally H_{l}.
- **gasUsed:** A scalar value equal to the total gas used in transactions in this block; formally H_{g}.
- **timestamp:** A scalar value equal to the reasonable output of Unix's time() at this block's inception; formally H_{s}.
- **extraData:** An arbitrary byte array containing data relevant to this block. This must be 32 bytes or fewer; formally H_{x}.
- **mixHash:** A 256-bit hash which, combined with the nonce, proves that a sufficient amount of computation has been carried out on this block; formally H_{m}.
- **nonce:** A 64-bit value which, combined with the mixhash, proves that a sufficient amount of computation has been carried out on this block; formally H_{n}.

The other two components in the block are simply a list of ommer block headers (of the same format as above), $B_{\mathbf{U}}$ and a series of the transactions, $B_{\mathbf{T}}$. Formally, we can refer to a block B:

$$B \equiv (B_H, B_{\mathbf{T}}, B_{\mathbf{U}}) \tag{19}$$

4.3.1. *Transaction Receipt.* In order to encode information about a transaction concerning which it may be useful to form a zero-knowledge proof, or index and search, we encode a receipt of each transaction containing certain information from its execution. Each receipt, denoted $B_{\mathbf{R}}[i]$ for the ith transaction, is placed in an index-keyed trie and the root recorded in the header as H_{e}.

The transaction receipt, R, is a tuple of four items comprising: the cumulative gas used in the block containing the transaction receipt as of immediately after the transaction has happened, R_{u}, the set of logs created through execution of the transaction, $R_{\mathbf{l}}$ and the Bloom filter composed from information in those logs, R_{b} and the status code of the transaction, R_{z}:

$$R \equiv (R_{\mathrm{u}}, R_{\mathrm{b}}, R_{\mathbf{l}}, R_z) \tag{20}$$

The function L_R trivially prepares a transaction receipt for being transformed into an RLP-serialised byte array:

$$L_R(R) \equiv (0 \in \mathbb{B}_{256}, R_{\mathrm{u}}, R_{\mathrm{b}}, R_{\mathbf{l}}) \tag{21}$$

where $0 \in \mathbb{B}_{256}$ replaces the pre-transaction state root that existed in previous versions of the protocol.

We assert that the status code R_{z} is a non-negative integer.

$$R_{\mathrm{z}} \in \mathbb{N} \tag{22}$$

We assert that R_{u}, the cumulative gas used, is a non-negative integer and that the logs Bloom, R_{b}, is a hash of size 2048 bits (256 bytes):

$$R_{\mathrm{u}} \in \mathbb{N} \quad \wedge \quad R_{\mathrm{b}} \in \mathbb{B}_{256} \tag{23}$$

The sequence $R_{\mathbf{l}}$ is a series of log entries, $(O_0, O_1, ...)$. A log entry, O, is a tuple of the logger's address, O_a, a possibly empty series of 32-byte log topics, $O_{\mathbf{t}}$ and some number of bytes of data, $O_{\mathbf{d}}$:

$$O \equiv (O_{\mathrm{a}}, (O_{\mathbf{t}0}, O_{\mathbf{t}1}, ...), O_{\mathbf{d}}) \tag{24}$$

$$O_{\mathrm{a}} \in \mathbb{B}_{20} \quad \wedge \quad \forall_{x \in O_{\mathbf{t}}} : x \in \mathbb{B}_{32} \quad \wedge \quad O_{\mathbf{d}} \in \mathbb{B} \tag{25}$$

We define the Bloom filter function, M, to reduce a log entry into a single 256-byte hash:

$$M(O) \equiv \bigvee_{x \in \{O_{\mathrm{a}}\} \cup O_{\mathbf{t}}} \left(M_{3:2048}(x) \right) \tag{26}$$

where $M_{3:2048}$ is a specialised Bloom filter that sets three bits out of 2048, given an arbitrary byte sequence. It does this through taking the low-order 11 bits of each of

[2] *ommer* is a gender-neutral term to mean "sibling of parent"; see `https://nonbinary.miraheze.org/wiki/Gender_neutral_language#Aunt.2FUncle`

[3] 11 bits = 2^{2048}, and the low-order 11 bits is the modulo 2048 of the operand, which is in this case is "each of the first three pairs of bytes in a Keccak-256 hash of the byte sequence."

the first three pairs of bytes in a Keccak-256 hash of the byte sequence.[3] Formally:

$$M_{3:2048}(\mathbf{x} : \mathbf{x} \in \mathbb{B}) \equiv \mathbf{y} : \mathbf{y} \in \mathbb{B}_{256} \quad \text{where:} \tag{27}$$
$$\mathbf{y} = (0, 0, ..., 0) \quad \text{except:} \tag{28}$$
$$\forall_{i \in \{0,2,4\}} : \mathcal{B}_{m(\mathbf{x},i)}(\mathbf{y}) = 1 \tag{29}$$
$$m(\mathbf{x}, i) \equiv \mathtt{KEC}(\mathbf{x})[i, i+1] \bmod 2048 \tag{30}$$

where $\mathcal{B}$ is the bit reference function such that $\mathcal{B}_{\mathrm{j}}(\mathbf{x})$ equals the bit of index j (indexed from 0) in the byte array $\mathbf{x}$.

4.3.2. *Holistic Validity.* We can assert a block's validity if and only if it satisfies several conditions: it must be internally consistent with the ommer and transaction block hashes and the given transactions $B_{\mathbf{T}}$ (as specified in sec 11), when executed in order on the base state $\boldsymbol{\sigma}$ (derived from the final state of the parent block), result in a new state of the identity H_{r}:

$$\begin{aligned} H_{\mathrm{r}} &\equiv \mathtt{TRIE}(L_S(\Pi(\boldsymbol{\sigma}, B))) && \wedge \\ H_{\mathrm{o}} &\equiv \mathtt{KEC}(\mathtt{RLP}(L^*_H(B_{\mathbf{U}}))) && \wedge \\ H_{\mathrm{t}} &\equiv \mathtt{TRIE}(\{\forall i < \|B_{\mathbf{T}}\|, i \in \mathbb{N} : \\ &\qquad p(i, L_T(B_{\mathbf{T}}[i]))\}) && \wedge \\ H_{\mathrm{e}} &\equiv \mathtt{TRIE}(\{\forall i < \|B_{\mathbf{R}}\|, i \in \mathbb{N} : \\ &\qquad p(i, L_R(B_{\mathbf{R}}[i]))\}) && \wedge \\ H_{\mathrm{b}} &\equiv \textstyle\bigvee_{\mathbf{r} \in B_{\mathbf{R}}} (\mathbf{r}_{\mathrm{b}}) \end{aligned} \tag{31}$$

where $p(k, v)$ is simply the pairwise RLP transformation, in this case, the first being the index of the transaction in the block and the second being the transaction receipt:

$$p(k, v) \equiv \big(\mathtt{RLP}(k), \mathtt{RLP}(v)\big) \tag{32}$$

Furthermore:

$$\mathtt{TRIE}(L_S(\boldsymbol{\sigma})) = P(B_H)_{H_{\mathrm{r}}} \tag{33}$$

Thus $\mathtt{TRIE}(L_S(\boldsymbol{\sigma}))$ is the root node hash of the Merkle Patricia tree structure containing the key-value pairs of the state $\boldsymbol{\sigma}$ with values encoded using RLP, and $P(B_H)$ is the parent block of B, defined directly.

The values stemming from the computation of transactions, specifically the transaction receipts, $B_{\mathbf{R}}$, and that defined through the transaction's state-accumulation function, Π, are formalised later in section 11.4.

4.3.3. *Serialisation.* The function L_B and L_H are the preparation functions for a block and block header respectively. Much like the transaction receipt preparation function L_R, we assert the types and order of the structure for when the RLP transformation is required:

$$\begin{aligned} L_H(H) &\equiv (\ H_{\mathrm{p}}, H_{\mathrm{o}}, H_{\mathrm{c}}, H_{\mathrm{r}}, H_{\mathrm{t}}, H_{\mathrm{e}}, H_{\mathrm{b}}, H_{\mathrm{d}}, \\ &\qquad H_{\mathrm{i}}, H_{\mathrm{l}}, H_{\mathrm{g}}, H_{\mathrm{s}}, H_{\mathrm{x}}, H_{\mathrm{m}}, H_{\mathrm{n}}\ ) \end{aligned} \tag{34}$$
$$L_B(B) \equiv \big(L_H(B_H), L^*_T(B_{\mathbf{T}}), L^*_H(B_{\mathbf{U}})\big) \tag{35}$$

With L^*_T and L^*_H being element-wise sequence transformations, thus:

$$f^*\big((x_0, x_1, ...)\big) \equiv \big(f(x_0), f(x_1), ...\big) \quad \text{for any function } f \tag{36}$$

The component types are defined thus:

$$\begin{aligned} &H_{\mathrm{p}} \in \mathbb{B}_{32} && \wedge && H_{\mathrm{o}} \in \mathbb{B}_{32} && \wedge && H_{\mathrm{c}} \in \mathbb{B}_{20} && \wedge \\ &H_{\mathrm{r}} \in \mathbb{B}_{32} && \wedge && H_{\mathrm{t}} \in \mathbb{B}_{32} && \wedge && H_{\mathrm{e}} \in \mathbb{B}_{32} && \wedge \\ &H_{\mathrm{b}} \in \mathbb{B}_{256} && \wedge && H_{\mathrm{d}} \in \mathbb{N} && \wedge && H_{\mathrm{i}} \in \mathbb{N} && \wedge \\ &H_{\mathrm{l}} \in \mathbb{N} && \wedge && H_{\mathrm{g}} \in \mathbb{N} && \wedge && H_{\mathrm{s}} \in \mathbb{N}_{256} && \wedge \\ &H_{\mathrm{x}} \in \mathbb{B} && \wedge && H_{\mathrm{m}} \in \mathbb{B}_{32} && \wedge && H_{\mathrm{n}} \in \mathbb{B}_{8} \end{aligned} \tag{37}$$

where

$$\mathbb{B}_{\mathrm{n}} = \{B : B \in \mathbb{B} \wedge \|B\| = n\} \tag{38}$$

We now have a rigorous specification for the construction of a formal block structure. The RLP function $\mathtt{RLP}$ (see Appendix B) provides the canonical method for transforming this structure into a sequence of bytes ready for transmission over the wire or storage locally.

4.3.4. *Block Header Validity.* We define $P(B_H)$ to be the parent block of B, formally:

$$P(H) \equiv B' : \mathtt{KEC}(\mathtt{RLP}(B'_H)) = H_{\mathrm{p}} \tag{39}$$

The block number is the parent's block number incremented by one:

$$H_{\mathrm{i}} \equiv P(H)_{H_{\mathrm{i}}} + 1 \tag{40}$$

The canonical difficulty of a block of header H is defined as $D(H)$:

$$D(H) \equiv \begin{cases} D_0 & \text{if} \quad H_{\mathrm{i}} = 0 \\ \max\big(D_0, P(H)_{H_{\mathrm{d}}} + x \times \varsigma_2 + \epsilon\big) & \text{otherwise} \end{cases} \tag{41}$$

where:

$$D_0 \equiv 131072 \tag{42}$$

$$x \equiv \left\lfloor \frac{P(H)_{H_{\mathrm{d}}}}{2048} \right\rfloor \tag{43}$$

$$\varsigma_2 \equiv \max\left(y - \left\lfloor \frac{H_{\mathrm{s}} - P(H)_{H_{\mathrm{s}}}}{9} \right\rfloor, -99\right) \tag{44}$$

$$y \equiv \begin{cases} 1 & \text{if } \|P(H)_{\mathbf{U}}\| = 0 \\ 2 & \text{otherwise} \end{cases}$$

$$\epsilon \equiv \left\lfloor 2^{\left\lfloor H'_{\mathrm{i}} \div 100000 \right\rfloor - 2} \right\rfloor \tag{45}$$
$$H'_{\mathrm{i}} \equiv \max(H_{\mathrm{i}} - 3000000, 0) \tag{46}$$

Note that D_0 is the difficulty of the genesis block. The *Homestead* difficulty parameter, ς_2, is used to affect a dynamic homeostasis of time between blocks, as the time between blocks varies, as discussed below, as implemented in EIP-2 by Buterin [2015]. In the Homestead release, the exponential difficulty symbol, ϵ causes the difficulty to slowly increase (every 100,000 blocks) at an exponential rate, and thus increasing the block time difference, and putting time pressure on transitioning to proof-of-stake. This effect, known as the "difficulty bomb", or "ice age", was explained in EIP-649 by Schoedon and Buterin [2017] and delayed and implemented earlier in EIP-2. ς_2 was also modified in EIP-100 with the use of x, the adjustment factor above, and the denominator 9, in order to target the mean block time including uncle blocks by Buterin [2016]. Finally, in the Byzantium release, with EIP-649, the ice age was delayed by creating a fake block number, H'_{i}, which is obtained by subtracting three million from the actual block number, which in other words reduced ϵ and the time difference between blocks, in order to allow

more time to develop proof-of-stake and preventing the network from "freezing" up.

The canonical gas limit H_l of a block of header H must fulfil the relation:

$$H_\mathrm{l} < P(H)_{H_\mathrm{l}} + \left\lfloor \frac{P(H)_{H_\mathrm{l}}}{1024} \right\rfloor \quad \wedge \quad H_\mathrm{l} > P(H)_{H_\mathrm{l}} - \left\lfloor \frac{P(H)_{H_\mathrm{l}}}{1024} \right\rfloor \quad \wedge \quad H_\mathrm{l} \geqslant 5000 \tag{47}$$

H_s is the timestamp (in Unix's time()) of block H and must fulfil the relation:

$$H_\mathrm{s} > P(H)_{H_\mathrm{s}} \tag{48}$$

This mechanism enforces a homeostasis in terms of the time between blocks; a smaller period between the last two blocks results in an increase in the difficulty level and thus additional computation required, lengthening the likely next period. Conversely, if the period is too large, the difficulty, and expected time to the next block, is reduced.

The nonce, H_n, must satisfy the relations:

$$n \leqslant \frac{2^{256}}{H_\mathrm{d}} \quad \wedge \quad m = H_\mathrm{m} \tag{49}$$

with $(n, m) = \mathtt{PoW}(H_{\not{N}}, H_\mathrm{n}, \mathbf{d})$.

Where $H_{\not{N}}$ is the new block's header H, but *without* the nonce and mix-hash components, $\mathbf{d}$ being the current DAG, a large data set needed to compute the mix-hash, and $\mathtt{PoW}$ is the proof-of-work function (see section 11.5): this evaluates to an array with the first item being the mix-hash, to prove that a correct DAG has been used, and the second item being a pseudo-random number cryptographically dependent on H and $\mathbf{d}$. Given an approximately uniform distribution in the range $[0, 2^{64})$, the expected time to find a solution is proportional to the difficulty, H_d.

This is the foundation of the security of the blockchain and is the fundamental reason why a malicious node cannot propagate newly created blocks that would otherwise overwrite ("rewrite") history. Because the nonce must satisfy this requirement, and because its satisfaction depends on the contents of the block and in turn its composed transactions, creating new, valid, blocks is difficult and, over time, requires approximately the total compute power of the trustworthy portion of the mining peers.

Thus we are able to define the block header validity function $V(H)$:

$$\begin{aligned} V(H) \equiv\ & n \leqslant \frac{2^{256}}{H_\mathrm{d}} \wedge m = H_\mathrm{m} \quad \wedge \\ & H_\mathrm{d} = D(H) \quad \wedge \\ & H_\mathrm{g} \leq H_\mathrm{l} \quad \wedge \\ & H_\mathrm{l} < P(H)_{H_\mathrm{l}} + \left\lfloor \frac{P(H)_{H_\mathrm{l}}}{1024} \right\rfloor \quad \wedge \\ & H_\mathrm{l} > P(H)_{H_\mathrm{l}} - \left\lfloor \frac{P(H)_{H_\mathrm{l}}}{1024} \right\rfloor \quad \wedge \\ & H_\mathrm{l} \geqslant 5000 \quad \wedge \\ & H_\mathrm{s} > P(H)_{H_\mathrm{s}} \quad \wedge \\ & H_\mathrm{i} = P(H)_{H_\mathrm{i}} + 1 \quad \wedge \\ & \|H_\mathrm{x}\| \leq 32 \end{aligned} \tag{50}$$

where $(n, m) = \mathtt{PoW}(H_{\not{N}}, H_\mathrm{n}, \mathbf{d})$

Noting additionally that **extraData** must be at most 32 bytes.

5. Gas and Payment

In order to avoid issues of network abuse and to sidestep the inevitable questions stemming from Turing completeness, all programmable computation in Ethereum is subject to fees. The fee schedule is specified in units of *gas* (see Appendix G for the fees associated with various computation). Thus any given fragment of programmable computation (this includes creating contracts, making message calls, utilising and accessing account storage and executing operations on the virtual machine) has a universally agreed cost in terms of gas.

Every transaction has a specific amount of gas associated with it: **gasLimit**. This is the amount of gas which is implicitly purchased from the sender's account balance. The purchase happens at the according **gasPrice**, also specified in the transaction. The transaction is considered invalid if the account balance cannot support such a purchase. It is named **gasLimit** since any unused gas at the end of the transaction is refunded (at the same rate of purchase) to the sender's account. Gas does not exist outside of the execution of a transaction. Thus for accounts with trusted code associated, a relatively high gas limit may be set and left alone.

In general, Ether used to purchase gas that is not refunded is delivered to the *beneficiary* address, the address of an account typically under the control of the miner. Transactors are free to specify any **gasPrice** that they wish, however miners are free to ignore transactions as they choose. A higher gas price on a transaction will therefore cost the sender more in terms of Ether and deliver a greater value to the miner and thus will more likely be selected for inclusion by more miners. Miners, in general, will choose to advertise the minimum gas price for which they will execute transactions and transactors will be free to canvas these prices in determining what gas price to offer. Since there will be a (weighted) distribution of minimum acceptable gas prices, transactors will necessarily have a trade-off to make between lowering the gas price and maximising the chance that their transaction will be mined in a timely manner.

6. Transaction Execution

The execution of a transaction is the most complex part of the Ethereum protocol: it defines the state transition function Υ. It is assumed that any transactions executed first pass the initial tests of intrinsic validity. These include:

(1) The transaction is well-formed RLP, with no additional trailing bytes;
(2) the transaction signature is valid;
(3) the transaction nonce is valid (equivalent to the sender account's current nonce);
(4) the gas limit is no smaller than the intrinsic gas, g_0, used by the transaction; and
(5) the sender account balance contains at least the cost, v_0, required in up-front payment.

Formally, we consider the function Υ, with T being a transaction and $\boldsymbol{\sigma}$ the state:

$$\boldsymbol{\sigma}' = \Upsilon(\boldsymbol{\sigma}, T) \tag{51}$$

Thus $\boldsymbol{\sigma}'$ is the post-transactional state. We also define Υ^g to evaluate to the amount of gas used in the execution of a transaction, Υ^l to evaluate to the transaction's accrued

log items and Υ^{z} to evaluate to the status code resulting from the transaction. These will be formally defined later.

6.1. **Substate.** Throughout transaction execution, we accrue certain information that is acted upon immediately following the transaction. We call this *transaction substate*, and represent it as A, which is a tuple:

$$A \equiv (A_{\mathbf{s}}, A_{\mathbf{l}}, A_{\mathbf{t}}, A_{\mathrm{r}}) \tag{52}$$

The tuple contents include $A_{\mathbf{s}}$, the self-destruct set: a set of accounts that will be discarded following the transaction's completion. $A_{\mathbf{l}}$ is the log series: this is a series of archived and indexable 'checkpoints' in VM code execution that allow for contract-calls to be easily tracked by onlookers external to the Ethereum world (such as decentralised application front-ends). $A_{\mathbf{t}}$ is the set of touched accounts, of which the empty ones are deleted at the end of a transaction. Finally there is A_{r}, the refund balance, increased through using the SSTORE instruction in order to reset contract storage to zero from some non-zero value. Though not immediately refunded, it is allowed to partially offset the total execution costs.

We define the empty substate A^0 to have no self-destructs, no logs, no touched accounts and a zero refund balance:

$$A^0 \equiv (\varnothing, (), \varnothing, 0) \tag{53}$$

6.2. **Execution.** We define intrinsic gas g_0, the amount of gas this transaction requires to be paid prior to execution, as follows:

$$g_0 \equiv \sum_{i \in T_{\mathbf{i}}, T_{\mathbf{d}}} \begin{cases} G_{\text{txdatazero}} & \text{if} \quad i = 0 \\ G_{\text{txdatanonzero}} & \text{otherwise} \end{cases} \tag{54}$$

$$+ \begin{cases} G_{\text{txcreate}} & \text{if} \quad T_{\mathrm{t}} = \varnothing \\ 0 & \text{otherwise} \end{cases} \tag{55}$$

$$+ G_{\text{transaction}} \tag{56}$$

where $T_{\mathbf{i}}, T_{\mathbf{d}}$ means the series of bytes of the transaction's associated data and initialisation EVM-code, depending on whether the transaction is for contract-creation or message-call. G_{txcreate} is added if the transaction is contract-creating, but not if a result of EVM-code. G is fully defined in Appendix G.

The up-front cost v_0 is calculated as:

$$v_0 \equiv T_{\mathrm{g}} T_{\mathrm{p}} + T_{\mathrm{v}} \tag{57}$$

The validity is determined as:

$$\begin{aligned} S(T) &\neq \varnothing \quad \wedge \\ \boldsymbol{\sigma}[S(T)] &\neq \varnothing \quad \wedge \\ T_{\mathrm{n}} &= \boldsymbol{\sigma}[S(T)]_{\mathrm{n}} \quad \wedge \\ g_0 &\leqslant T_{\mathrm{g}} \quad \wedge \\ v_0 &\leqslant \boldsymbol{\sigma}[S(T)]_{\mathrm{b}} \quad \wedge \\ T_{\mathrm{g}} &\leqslant B_{H\mathrm{l}} - \ell(B_{\mathbf{R}})_{\mathrm{u}} \end{aligned} \tag{58}$$

Note the final condition; the sum of the transaction's gas limit, T_{g}, and the gas utilised in this block prior, given by $\ell(B_{\mathbf{R}})_{\mathrm{u}}$, must be no greater than the block's **gasLimit**, $B_{H\mathrm{l}}$.

The execution of a valid transaction begins with an irrevocable change made to the state: the nonce of the account of the sender, $S(T)$, is incremented by one and the balance is reduced by part of the up-front cost, $T_{\mathrm{g}} T_{\mathrm{p}}$. The gas available for the proceeding computation, g, is defined as $T_{\mathrm{g}} - g_0$. The computation, whether contract creation or a message call, results in an eventual state (which may legally be equivalent to the current state), the change to which is deterministic and never invalid: there can be no invalid transactions from this point.

We define the checkpoint state $\boldsymbol{\sigma}_0$:

$$\boldsymbol{\sigma}_0 \equiv \boldsymbol{\sigma} \quad \text{except:} \tag{59}$$

$$\boldsymbol{\sigma}_0[S(T)]_{\mathrm{b}} \equiv \boldsymbol{\sigma}[S(T)]_{\mathrm{b}} - T_{\mathrm{g}} T_{\mathrm{p}} \tag{60}$$

$$\boldsymbol{\sigma}_0[S(T)]_{\mathrm{n}} \equiv \boldsymbol{\sigma}[S(T)]_{\mathrm{n}} + 1 \tag{61}$$

Evaluating $\boldsymbol{\sigma}_P$ from $\boldsymbol{\sigma}_0$ depends on the transaction type; either contract creation or message call; we define the tuple of post-execution provisional state $\boldsymbol{\sigma}_P$, remaining gas g', substate A and status code z:

$$(\boldsymbol{\sigma}_P, g', A, z) \equiv \begin{cases} \Lambda_4(\boldsymbol{\sigma}_0, S(T), T_{\mathrm{o}}, g, \\ \quad T_{\mathrm{p}}, T_{\mathrm{v}}, T_{\mathbf{i}}, 0, \top) & \text{if} \quad T_{\mathrm{t}} = \varnothing \\ \Theta_4(\boldsymbol{\sigma}_0, S(T), T_{\mathrm{o}}, T_{\mathrm{t}}, T_{\mathrm{t}}, \\ \quad g, T_{\mathrm{p}}, T_{\mathrm{v}}, T_{\mathbf{d}}, 0, \top) & \text{otherwise} \end{cases} \tag{62}$$

where g is the amount of gas remaining after deducting the basic amount required to pay for the existence of the transaction:

$$g \equiv T_{\mathrm{g}} - g_0 \tag{63}$$

and T_{o} is the original transactor, which can differ from the sender in the case of a message call or contract creation not directly triggered by a transaction but coming from the execution of EVM-code.

Note we use Θ_4 and Λ_4 to denote the fact that only the first four components of the functions' values are taken; the final represents the message-call's output value (a byte array) and is unused in the context of transaction evaluation.

After the message call or contract creation is processed, the refund counter has to be incremented for the accounts that were self-destructed throughout its invocation.

$$A'_r \equiv A_r + \sum_{i \in A_s} R_{selfdestruct} \tag{64}$$

Then the state is finalised by determining the amount to be refunded, g^* from the remaining gas, g', plus some allowance from the refund counter, to the sender at the original rate.

$$g^* \equiv g' + \min\left\{ \left\lfloor \frac{T_{\mathrm{g}} - g'}{2} \right\rfloor, A'_{\mathrm{r}} \right\} \tag{65}$$

The total refundable amount is the legitimately remaining gas g', added to A_{r}, with the latter component being capped up to a maximum of half (rounded down) of the total amount used $T_{\mathrm{g}} - g'$. Therefore, g^* is the total gas that remains after the transaction has been executed.

The Ether for the gas is given to the miner, whose address is specified as the beneficiary of the present block B. So we define the pre-final state $\boldsymbol{\sigma}^*$ in terms of the provisional state $\boldsymbol{\sigma}_P$:

$$\boldsymbol{\sigma}^* \equiv \boldsymbol{\sigma}_P \quad \text{except} \tag{66}$$

$$\boldsymbol{\sigma}^*[S(T)]_{\mathrm{b}} \equiv \boldsymbol{\sigma}_P[S(T)]_{\mathrm{b}} + g^* T_{\mathrm{p}} \tag{67}$$

$$\boldsymbol{\sigma}^*[m]_{\mathrm{b}} \equiv \boldsymbol{\sigma}_P[m]_{\mathrm{b}} + (T_{\mathrm{g}} - g^*) T_{\mathrm{p}} \tag{68}$$

$$m \equiv B_{H\mathrm{c}} \tag{69}$$

The final state, $\boldsymbol{\sigma}'$, is reached after deleting all accounts that either appear in the self-destruct list or are touched

and empty:

$$\boldsymbol{\sigma}' \equiv \boldsymbol{\sigma}^* \quad \text{except} \tag{70}$$

$$\forall i \in A_{\mathbf{s}} : \boldsymbol{\sigma}'[i] = \varnothing \tag{71}$$

$$\forall i \in A_{\mathbf{t}} : \boldsymbol{\sigma}'[i] = \varnothing \quad \text{if} \quad \texttt{DEAD}(\boldsymbol{\sigma}^*, i) \tag{72}$$

And finally, we specify Υ^{g}, the total gas used in this transaction $\Upsilon^{\mathbf{l}}$, the logs created by this transaction and Υ^{z}, the status code of this transaction:

$$\Upsilon^{\mathrm{g}}(\boldsymbol{\sigma}, T) \equiv T_{\mathrm{g}} - g^* \tag{73}$$

$$\Upsilon^{\mathbf{l}}(\boldsymbol{\sigma}, T) \equiv A_{\mathbf{l}} \tag{74}$$

$$\Upsilon^{\mathrm{z}}(\boldsymbol{\sigma}, T) \equiv z \tag{75}$$

These are used to help define the transaction receipt and are also used later for state and nonce validation.

7. Contract Creation

There are a number of intrinsic parameters used when creating an account: sender (s), original transactor (o), available gas (g), gas price (p), endowment (v) together with an arbitrary length byte array, $\mathbf{i}$, the initialisation EVM code, the present depth of the message-call/contract-creation stack (e) and finally the permission to make modifications to the state (w).

We define the creation function formally as the function Λ, which evaluates from these values, together with the state $\boldsymbol{\sigma}$ to the tuple containing the new state, remaining gas, accrued transaction substate and an error message $(\boldsymbol{\sigma}', g', A, \mathbf{o})$, as in section 6:

$$(\boldsymbol{\sigma}', g', A, z, \mathbf{o}) \equiv \Lambda(\boldsymbol{\sigma}, s, o, g, p, v, \mathbf{i}, e, w) \tag{76}$$

The address of the new account is defined as being the rightmost 160 bits of the Keccak hash of the RLP encoding of the structure containing only the sender and the account nonce. Thus we define the resultant address for the new account a:

$$a \equiv \mathcal{B}_{96..255}\Big(\texttt{KEC}\Big(\texttt{RLP}\big(\ (s, \boldsymbol{\sigma}[s]_{\mathrm{n}} - 1)\ \big)\Big)\Big) \tag{77}$$

where $\texttt{KEC}$ is the Keccak 256-bit hash function, $\texttt{RLP}$ is the RLP encoding function, $\mathcal{B}_{a..b}(X)$ evaluates to a binary value containing the bits of indices in the range $[a, b]$ of the binary data X, and $\boldsymbol{\sigma}[x]$ is the address state of x, or $\varnothing$ if none exists. Note we use one fewer than the sender's nonce value; we assert that we have incremented the sender account's nonce prior to this call, and so the value used is the sender's nonce at the beginning of the responsible transaction or VM operation.

The account's nonce is initially defined as one, the balance as the value passed, the storage as empty and the code hash as the Keccak 256-bit hash of the empty string; the sender's balance is also reduced by the value passed. Thus the mutated state becomes $\boldsymbol{\sigma}^*$:

$$\boldsymbol{\sigma}^* \equiv \boldsymbol{\sigma} \quad \text{except:} \tag{78}$$

$$\boldsymbol{\sigma}^*[a] = \big(1, v + v', \texttt{TRIE}(\varnothing), \texttt{KEC}\big(()\big)\big) \tag{79}$$

$$\boldsymbol{\sigma}^*[s] = \begin{cases} \varnothing & \text{if } \boldsymbol{\sigma}[s] = \varnothing \ \wedge \ v = 0 \\ \mathbf{a}^* & \text{otherwise} \end{cases} \tag{80}$$

$$\mathbf{a}^* \equiv (\boldsymbol{\sigma}[s]_{\mathrm{n}}, \boldsymbol{\sigma}[s]_{\mathrm{b}} - v, \boldsymbol{\sigma}[s]_{\mathbf{s}}, \boldsymbol{\sigma}[s]_{\mathrm{c}}) \tag{81}$$

where v' is the account's pre-existing value, in the event it was previously in existence:

$$v' \equiv \begin{cases} 0 & \text{if} \quad \boldsymbol{\sigma}[a] = \varnothing \\ \boldsymbol{\sigma}[a]_{\mathrm{b}} & \text{otherwise} \end{cases} \tag{82}$$

Finally, the account is initialised through the execution of the initialising EVM code $\mathbf{i}$ according to the execution model (see section 9). Code execution can effect several events that are not internal to the execution state: the account's storage can be altered, further accounts can be created and further message calls can be made. As such, the code execution function Ξ evaluates to a tuple of the resultant state $\boldsymbol{\sigma}^{**}$, available gas remaining g^{**}, the accrued substate A and the body code of the account $\mathbf{o}$.

$$(\boldsymbol{\sigma}^{**}, g^{**}, A, \mathbf{o}) \equiv \Xi(\boldsymbol{\sigma}^*, g, I, \{s, a\}) \tag{83}$$

where I contains the parameters of the execution environment, that is:

$$I_{\mathrm{a}} \equiv a \tag{84}$$

$$I_{\mathrm{o}} \equiv o \tag{85}$$

$$I_{\mathrm{p}} \equiv p \tag{86}$$

$$I_{\mathbf{d}} \equiv () \tag{87}$$

$$I_{\mathrm{s}} \equiv s \tag{88}$$

$$I_{\mathrm{v}} \equiv v \tag{89}$$

$$I_{\mathbf{b}} \equiv \mathbf{i} \tag{90}$$

$$I_{\mathrm{e}} \equiv e \tag{91}$$

$$I_{\mathrm{w}} \equiv w \tag{92}$$

$I_{\mathbf{d}}$ evaluates to the empty tuple as there is no input data to this call. I_H has no special treatment and is determined from the blockchain.

Code execution depletes gas, and gas may not go below zero, thus execution may exit before the code has come to a natural halting state. In this (and several other) exceptional cases we say an out-of-gas (OOG) exception has occurred: The evaluated state is defined as being the empty set, $\varnothing$, and the entire create operation should have no effect on the state, effectively leaving it as it was immediately prior to attempting the creation.

If the initialization code completes successfully, a final contract-creation cost is paid, the code-deposit cost, c, proportional to the size of the created contract's code:

$$c \equiv G_{codedeposit} \times |\mathbf{o}| \tag{93}$$

If there is not enough gas remaining to pay this, i.e. $g^{**} < c$, then we also declare an out-of-gas exception.

The gas remaining will be zero in any such exceptional condition, i.e. if the creation was conducted as the reception of a transaction, then this doesn't affect payment of the intrinsic cost of contract creation; it is paid regardless. However, the value of the transaction is not transferred to the aborted contract's address when we are out-of-gas.

If such an exception does not occur, then the remaining gas is refunded to the originator and the now-altered state is allowed to persist. Thus formally, we may specify the resultant state, gas, substate and status code as $(\boldsymbol{\sigma}', g', A, z)$ where:

(94)

$$g' \equiv \begin{cases} 0 & \text{if} \quad F \\ g^{**} - c & \text{otherwise} \end{cases}$$

(95)

$$\boldsymbol{\sigma}' \equiv \begin{cases} \boldsymbol{\sigma} & \text{if} \quad F \\ \boldsymbol{\sigma}^{**} \quad \text{except:} & \\ \quad \boldsymbol{\sigma}'[a] = \varnothing & \text{if} \quad \texttt{DEAD}(\boldsymbol{\sigma}^{**}, a) \\ \boldsymbol{\sigma}^{**} \quad \text{except:} & \\ \quad \boldsymbol{\sigma}'[a]_{\mathrm{c}} = \texttt{KEC}(\mathbf{o}) & \text{otherwise} \end{cases}$$

(96)

$$z \equiv \begin{cases} 0 & \text{if} \quad \boldsymbol{\sigma}^{**} = \varnothing \vee g^{**} < c \\ 1 & \text{otherwise} \end{cases}$$

where

(97)

$$F \equiv \big((\boldsymbol{\sigma}^{**} = \varnothing \ \wedge \ \mathbf{o} = \varnothing) \vee \ g^{**} < c \ \vee \ |\mathbf{o}| > 24576\big)$$

The exception in the determination of $\boldsymbol{\sigma}'$ dictates that $\mathbf{o}$, the resultant byte sequence from the execution of the initialisation code, specifies the final body code for the newly-created account.

Note that intention is that the result is either a successfully created new contract with its endowment, or no new contract with no transfer of value.

7.1. **Subtleties.** Note that while the initialisation code is executing, the newly created address exists but with no intrinsic body code[4]. Thus any message call received by it during this time causes no code to be executed. If the initialisation execution ends with a SELFDESTRUCT instruction, the matter is moot since the account will be deleted before the transaction is completed. For a normal STOP code, or if the code returned is otherwise empty, then the state is left with a zombie account, and any remaining balance will be locked into the account forever.

8. Message Call

In the case of executing a message call, several parameters are required: sender (s), transaction originator (o), recipient (r), the account whose code is to be executed (c, usually the same as recipient), available gas (g), value (v) and gas price (p) together with an arbitrary length byte array, $\mathbf{d}$, the input data of the call, the present depth of the message-call/contract-creation stack (e) and finally the permission to make modifications to the state (w).

Aside from evaluating to a new state and transaction substate, message calls also have an extra component—the output data denoted by the byte array $\mathbf{o}$. This is ignored when executing transactions, however message calls can be initiated due to VM-code execution and in this case this information is used.

(98) $$(\boldsymbol{\sigma}', g', A, z, \mathbf{o}) \equiv \Theta(\boldsymbol{\sigma}, s, o, r, c, g, p, v, \tilde{v}, \mathbf{d}, e, w)$$

Note that we need to differentiate between the value that is to be transferred, v, from the value apparent in the execution context, $\tilde{v}$, for the DELEGATECALL instruction.

We define $\boldsymbol{\sigma}_1$, the first transitional state as the original state but with the value transferred from sender to recipient:

(99) $$\boldsymbol{\sigma}_1[r]_{\mathrm{b}} \equiv \boldsymbol{\sigma}[r]_{\mathrm{b}} + v \quad \wedge \quad \boldsymbol{\sigma}_1[s]_{\mathrm{b}} \equiv \boldsymbol{\sigma}[s]_{\mathrm{b}} - v$$

unless $s = r$.

Throughout the present work, it is assumed that if $\boldsymbol{\sigma}_1[r]$ was originally undefined, it will be created as an account with no code or state and zero balance and nonce. Thus the previous equation should be taken to mean:

(100) $$\boldsymbol{\sigma}_1 \equiv \boldsymbol{\sigma}'_1 \quad \text{except:}$$

(101) $$\boldsymbol{\sigma}_1[s] \equiv \begin{cases} \varnothing & \text{if } \boldsymbol{\sigma}'_1[s] = \varnothing \ \wedge \ v = 0 \\ \mathbf{a}_1 & \text{otherwise} \end{cases}$$

(102) $$\mathbf{a}_1 \equiv \big(\boldsymbol{\sigma}'_1[s]_{\mathrm{n}}, \boldsymbol{\sigma}'_1[s]_{\mathrm{b}} - v, \boldsymbol{\sigma}'_1[s]_{\mathbf{s}}, \boldsymbol{\sigma}'_1[s]_{\mathrm{c}}\big)$$

(103) $$\text{and} \quad \boldsymbol{\sigma}'_1 \equiv \boldsymbol{\sigma} \quad \text{except:}$$

(104)

$$\begin{cases} \boldsymbol{\sigma}'_1[r] \equiv (0, v, \texttt{TRIE}(\varnothing), \texttt{KEC}(())) & \text{if} \quad \boldsymbol{\sigma}[r] = \varnothing \wedge v \neq 0 \\ \boldsymbol{\sigma}'_1[r] \equiv \varnothing & \text{if} \quad \boldsymbol{\sigma}[r] = \varnothing \wedge v = 0 \\ \boldsymbol{\sigma}'_1[r] \equiv \mathbf{a}'_1 & \text{otherwise} \end{cases}$$

(105) $$\mathbf{a}'_1 \equiv (\boldsymbol{\sigma}[r]_{\mathrm{n}}, \boldsymbol{\sigma}[r]_{\mathrm{b}} + v, \boldsymbol{\sigma}[r]_{\mathbf{s}}, \boldsymbol{\sigma}[r]_{\mathrm{c}})$$

The account's associated code (identified as the fragment whose Keccak hash is $\boldsymbol{\sigma}[c]_{\mathrm{c}}$) is executed according to the execution model (see section 9). Just as with contract creation, if the execution halts in an exceptional fashion (i.e. due to an exhausted gas supply, stack underflow, invalid jump destination or invalid instruction), then no gas is refunded to the caller and the state is reverted to the point immediately prior to balance transfer (i.e. $\boldsymbol{\sigma}$).

(106) $$\boldsymbol{\sigma}' \equiv \begin{cases} \boldsymbol{\sigma} & \text{if} \quad \boldsymbol{\sigma}^{**} = \varnothing \\ \boldsymbol{\sigma}^{**} & \text{otherwise} \end{cases}$$

(107) $$g' \equiv \begin{cases} 0 & \text{if} \quad \boldsymbol{\sigma}^{**} = \varnothing \ \wedge \\ & \quad \mathbf{o} = \varnothing \\ g^{**} & \text{otherwise} \end{cases}$$

$$z \equiv \begin{cases} 0 & \text{if} \quad \boldsymbol{\sigma}^{**} = \varnothing \\ 1 & \text{otherwise} \end{cases}$$

(108) $$(\boldsymbol{\sigma}^{**}, g^{**}, A, \mathbf{o}) \equiv \Xi$$

(109) $$I_{\mathrm{a}} \equiv r$$

(110) $$I_{\mathrm{o}} \equiv o$$

(111) $$I_{\mathrm{p}} \equiv p$$

(112) $$I_{\mathbf{d}} \equiv \mathbf{d}$$

(113) $$I_{\mathrm{s}} \equiv s$$

(114) $$I_{\mathrm{v}} \equiv \tilde{v}$$

(115) $$I_{\mathrm{e}} \equiv e$$

(116) $$I_{\mathrm{w}} \equiv w$$

(117) $$\mathbf{t} \equiv \{s, r\}$$

(118)

[4]During initialization code execution, `EXTCODESIZE` on the address should return zero, which is the length of the code of the account while `CODESIZE` should return the length of the initialization code (as defined in H.2).

where

$$\Xi \equiv \begin{cases} \Xi_{\texttt{ECREC}}(\boldsymbol{\sigma}_1, g, I, \mathbf{t}) & \text{if} \quad r = 1 \\ \Xi_{\texttt{SHA256}}(\boldsymbol{\sigma}_1, g, I, \mathbf{t}) & \text{if} \quad r = 2 \\ \Xi_{\texttt{RIP160}}(\boldsymbol{\sigma}_1, g, I, \mathbf{t}) & \text{if} \quad r = 3 \\ \Xi_{\texttt{ID}}(\boldsymbol{\sigma}_1, g, I, \mathbf{t}) & \text{if} \quad r = 4 \\ \Xi_{\texttt{EXPMOD}}(\boldsymbol{\sigma}_1, g, I, \mathbf{t}) & \text{if} \quad r = 5 \\ \Xi_{\texttt{BN_ADD}}(\boldsymbol{\sigma}_1, g, I, \mathbf{t}) & \text{if} \quad r = 6 \\ \Xi_{\texttt{BN_MUL}}(\boldsymbol{\sigma}_1, g, I, \mathbf{t}) & \text{if} \quad r = 7 \\ \Xi_{\texttt{SNARKV}}(\boldsymbol{\sigma}_1, g, I, \mathbf{t}) & \text{if} \quad r = 8 \\ \Xi(\boldsymbol{\sigma}_1, g, I, \mathbf{t}) & \text{otherwise} \end{cases} \tag{119}$$

$$\text{Let } \texttt{KEC}(I_{\mathbf{b}}) = \boldsymbol{\sigma}[c]_{\mathrm{c}} \tag{120}$$

It is assumed that the client will have stored the pair $(\texttt{KEC}(I_{\mathbf{b}}), I_{\mathbf{b}})$ at some point prior in order to make the determination of $I_{\mathbf{b}}$ feasible.

As can be seen, there are eight exceptions to the usage of the general execution framework Ξ for evaluation of the message call: these are eight so-called 'precompiled' contracts, meant as a preliminary piece of architecture that may later become *native extensions*. The eight contracts in addresses 1 to 8 execute the elliptic curve public key recovery function, the SHA2 256-bit hash scheme, the RIPEMD 160-bit hash scheme, the identity function, arbitrary precision modular exponentiation, elliptic curve addition, elliptic curve scalar multiplication and an elliptic curve pairing check respectively.

Their full formal definition is in Appendix E.

9. Execution Model

The execution model specifies how the system state is altered given a series of bytecode instructions and a small tuple of environmental data. This is specified through a formal model of a virtual state machine, known as the Ethereum Virtual Machine (EVM). It is a *quasi*-Turing-complete machine; the *quasi* qualification comes from the fact that the computation is intrinsically bounded through a parameter, *gas*, which limits the total amount of computation done.

9.1. **Basics.** The EVM is a simple stack-based architecture. The word size of the machine (and thus size of stack items) is 256-bit. This was chosen to facilitate the Keccak-256 hash scheme and elliptic-curve computations. The memory model is a simple word-addressed byte array. The stack has a maximum size of 1024. The machine also has an independent storage model; this is similar in concept to the memory but rather than a byte array, it is a word-addressable word array. Unlike memory, which is volatile, storage is non volatile and is maintained as part of the system state. All locations in both storage and memory are well-defined initially as zero.

The machine does not follow the standard von Neumann architecture. Rather than storing program code in generally-accessible memory or storage, it is stored separately in a virtual ROM interactable only through a specialised instruction.

The machine can have exceptional execution for several reasons, including stack underflows and invalid instructions. Like the out-of-gas exception, they do not leave state changes intact. Rather, the machine halts immediately and reports the issue to the execution agent (either the transaction processor or, recursively, the spawning execution environment) which will deal with it separately.

9.2. **Fees Overview.** Fees (denominated in gas) are charged under three distinct circumstances, all three as prerequisite to the execution of an operation. The first and most common is the fee intrinsic to the computation of the operation (see Appendix G). Secondly, gas may be deducted in order to form the payment for a subordinate message call or contract creation; this forms part of the payment for CREATE, CALL and CALLCODE. Finally, gas may be paid due to an increase in the usage of the memory.

Over an account's execution, the total fee for memory-usage payable is proportional to smallest multiple of 32 bytes that are required such that all memory indices (whether for read or write) are included in the range. This is paid for on a just-in-time basis; as such, referencing an area of memory at least 32 bytes greater than any previously indexed memory will certainly result in an additional memory usage fee. Due to this fee it is highly unlikely addresses will ever go above 32-bit bounds. That said, implementations must be able to manage this eventuality.

Storage fees have a slightly nuanced behaviour—to incentivise minimisation of the use of storage (which corresponds directly to a larger state database on all nodes), the execution fee for an operation that clears an entry in the storage is not only waived, a qualified refund is given; in fact, this refund is effectively paid up-front since the initial usage of a storage location costs substantially more than normal usage.

See Appendix H for a rigorous definition of the EVM gas cost.

9.3. **Execution Environment.** In addition to the system state $\boldsymbol{\sigma}$, and the remaining gas for computation g, there are several pieces of important information used in the execution environment that the execution agent must provide; these are contained in the tuple I:

- I_{a}, the address of the account which owns the code that is executing.
- I_{o}, the sender address of the transaction that originated this execution.
- I_{p}, the price of gas in the transaction that originated this execution.
- $I_{\mathbf{d}}$, the byte array that is the input data to this execution; if the execution agent is a transaction, this would be the transaction data.
- I_{s}, the address of the account which caused the code to be executing; if the execution agent is a transaction, this would be the transaction sender.
- I_{v}, the value, in Wei, passed to this account as part of the same procedure as execution; if the execution agent is a transaction, this would be the transaction value.
- $I_{\mathbf{b}}$, the byte array that is the machine code to be executed.
- $I_{\mathbf{H}}$, the block header of the present block.
- I_{e}, the depth of the present message-call or contract-creation (i.e. the number of CALLs or CREATEs being executed at present).
- I_{w}, the permission to make modifications to the state.

The execution model defines the function Ξ, which can compute the resultant state $\boldsymbol{\sigma}'$, the remaining gas g', the

accrued substate A and the resultant output, $\mathbf{o}$, given these definitions. For the present context, we will define it as:

$$(\boldsymbol{\sigma}', g', A, \mathbf{o}) \equiv \Xi(\boldsymbol{\sigma}, g, I) \tag{121}$$

where we will remember that A, the accrued substate is defined as the tuple of the suicides set $\mathbf{s}$, the log series $\mathbf{l}$, the touched accounts $\mathbf{t}$ and the refunds r:

$$A \equiv (\mathbf{s}, \mathbf{l}, \mathbf{t}, r) \tag{122}$$

9.4. **Execution Overview.** We must now define the Ξ function. In most practical implementations this will be modelled as an iterative progression of the pair comprising the full system state, $\boldsymbol{\sigma}$ and the machine state, $\boldsymbol{\mu}$. Formally, we define it recursively with a function X. This uses an iterator function O (which defines the result of a single cycle of the state machine) together with functions Z which determines if the present state is an exceptional halting state of the machine and H, specifying the output data of the instruction if and only if the present state is a normal halting state of the machine.

The empty sequence, denoted (), is not equal to the empty set, denoted $\varnothing$; this is important when interpreting the output of H, which evaluates to $\varnothing$ when execution is to continue but a series (potentially empty) when execution should halt.

$$\Xi(\boldsymbol{\sigma}, g, I, T) \equiv (\boldsymbol{\sigma}', \boldsymbol{\mu}'_{\mathrm{g}}, A, \mathbf{o}) \tag{123}$$

$$(\boldsymbol{\sigma}', \boldsymbol{\mu}', A, ..., \mathbf{o}) \equiv X\big((\boldsymbol{\sigma}, \boldsymbol{\mu}, A^0, I)\big) \tag{124}$$

$$\boldsymbol{\mu}_{\mathrm{g}} \equiv g \tag{125}$$

$$\boldsymbol{\mu}_{\mathrm{pc}} \equiv 0 \tag{126}$$

$$\boldsymbol{\mu}_{\mathbf{m}} \equiv (0, 0, ...) \tag{127}$$

$$\boldsymbol{\mu}_{\mathrm{i}} \equiv 0 \tag{128}$$

$$\boldsymbol{\mu}_{\mathbf{s}} \equiv () \tag{129}$$

$$\boldsymbol{\mu}_{\mathbf{o}} \equiv () \tag{130}$$

(131)

$$X\big((\boldsymbol{\sigma}, \boldsymbol{\mu}, A, I)\big) \equiv \begin{cases} \big(\varnothing, \boldsymbol{\mu}, A^0, I, \varnothing\big) & \text{if} \quad Z(\boldsymbol{\sigma}, \boldsymbol{\mu}, I) \\ \big(\varnothing, \boldsymbol{\mu}', A^0, I, \mathbf{o}\big) & \text{if} \quad w = \text{REVERT} \\ O(\boldsymbol{\sigma}, \boldsymbol{\mu}, A, I) \cdot \mathbf{o} & \text{if} \quad \mathbf{o} \neq \varnothing \\ X\big(O(\boldsymbol{\sigma}, \boldsymbol{\mu}, A, I)\big) & \text{otherwise} \end{cases}$$

where

$$\mathbf{o} \equiv H(\boldsymbol{\mu}, I) \tag{132}$$

$$(a, b, c, d) \cdot e \equiv (a, b, c, d, e) \tag{133}$$

$$\boldsymbol{\mu}' \equiv \boldsymbol{\mu} \text{ except:} \tag{134}$$

$$\boldsymbol{\mu}'_{\mathrm{g}} \equiv \boldsymbol{\mu}_{\mathrm{g}} - C(\boldsymbol{\sigma}, \boldsymbol{\mu}, I) \tag{135}$$

Note that, when we evaluate Ξ, we drop the fourth element I' and extract the remaining gas $\boldsymbol{\mu}'_{\mathrm{g}}$ from the resultant machine state $\boldsymbol{\mu}'$.

X is thus cycled (recursively here, but implementations are generally expected to use a simple iterative loop) until either Z becomes true indicating that the present state is exceptional and that the machine must be halted and any changes discarded or until H becomes a series (rather than the empty set) indicating that the machine has reached a controlled halt.

9.4.1. *Machine State.* The machine state $\boldsymbol{\mu}$ is defined as the tuple $(g, pc, \mathbf{m}, i, \mathbf{s})$ which are the gas available, the program counter $pc \in \mathbb{N}_{256}$, the memory contents, the active number of words in memory (counting continuously from position 0), and the stack contents. The memory contents $\boldsymbol{\mu}_{\mathbf{m}}$ are a series of zeroes of size 2^{256}.

For the ease of reading, the instruction mnemonics, written in small-caps (e.g. ADD), should be interpreted as their numeric equivalents; the full table of instructions and their specifics is given in Appendix H.

For the purposes of defining Z, H and O, we define w as the current operation to be executed:

$$w \equiv \begin{cases} I_{\mathbf{b}}[\boldsymbol{\mu}_{\mathrm{pc}}] & \text{if} \quad \boldsymbol{\mu}_{\mathrm{pc}} < \|I_{\mathbf{b}}\| \\ \text{STOP} & \text{otherwise} \end{cases} \tag{136}$$

We also assume the fixed amounts of δ and α, specifying the stack items removed and added, both subscriptable on the instruction and an instruction cost function C evaluating to the full cost, in gas, of executing the given instruction.

9.4.2. *Exceptional Halting.* The exceptional halting function Z is defined as:

(137)

$$\begin{aligned} Z(\boldsymbol{\sigma}, \boldsymbol{\mu}, I) \equiv \; & \boldsymbol{\mu}_g < C(\boldsymbol{\sigma}, \boldsymbol{\mu}, I) \quad \vee \\ & \delta_w = \varnothing \quad \vee \\ & \|\boldsymbol{\mu}_{\mathbf{s}}\| < \delta_w \quad \vee \\ & (w = \text{JUMP} \quad \wedge \quad \boldsymbol{\mu}_{\mathbf{s}}[0] \notin D(I_{\mathbf{b}})) \quad \vee \\ & (w = \text{JUMPI} \quad \wedge \quad \boldsymbol{\mu}_{\mathbf{s}}[1] \neq 0 \quad \wedge \\ & \quad \boldsymbol{\mu}_{\mathbf{s}}[0] \notin D(I_{\mathbf{b}})) \quad \vee \\ & (w = \text{RETURNDATACOPY} \wedge \\ & \quad \boldsymbol{\mu}_{\mathbf{s}}[1] + \boldsymbol{\mu}_{\mathbf{s}}[2] > \|\boldsymbol{\mu}_{\mathbf{o}}\|) \quad \vee \\ & \|\boldsymbol{\mu}_{\mathbf{s}}\| - \delta_w + \alpha_w > 1024 \vee (\neg I_{\mathrm{w}} \wedge W(w, \boldsymbol{\mu})) \end{aligned}$$

where

(138)

$$\begin{aligned} W(w, \boldsymbol{\mu}) \equiv \; & w \in \{\text{CREATE}, \text{SSTORE}, \\ & \text{SELFDESTRUCT}\} \vee \\ & \text{LOG0} \leq w \wedge w \leq \text{LOG4} \quad \vee \\ & w \in \{\text{CALL}, \text{CALLCODE}\} \wedge \boldsymbol{\mu}_{\mathbf{s}}[2] \neq 0 \end{aligned}$$

This states that the execution is in an exceptional halting state if there is insufficient gas, if the instruction is invalid (and therefore its δ subscript is undefined), if there are insufficient stack items, if a JUMP/JUMPI destination is invalid, the new stack size would be larger than 1024 or state modification is attempted during a static call. The astute reader will realise that this implies that no instruction can, through its execution, cause an exceptional halt.

9.4.3. *Jump Destination Validity.* We previously used D as the function to determine the set of valid jump destinations given the code that is being run. We define this as any position in the code occupied by a JUMPDEST instruction.

All such positions must be on valid instruction boundaries, rather than sitting in the data portion of PUSH operations and must appear within the explicitly defined portion of the code (rather than in the implicitly defined STOP operations that trail it).

Formally:

$$D(\mathbf{c}) \equiv D_J(\mathbf{c}, 0) \tag{139}$$

where:

(140)
$$D_J(\mathbf{c}, i) \equiv \begin{cases} \{\} & \text{if} \quad i \geqslant |\mathbf{c}| \\ \{i\} \cup D_J(\mathbf{c}, N(i, \mathbf{c}[i])) & \\ \quad \text{if} \quad \mathbf{c}[i] = \text{JUMPDEST} & \\ D_J(\mathbf{c}, N(i, \mathbf{c}[i])) & \text{otherwise} \end{cases}$$

where N is the next valid instruction position in the code, skipping the data of a PUSH instruction, if any:

(141)
$$N(i, w) \equiv \begin{cases} i + w - \text{PUSH1} + 2 & \\ \quad \text{if} \quad w \in [\text{PUSH1}, \text{PUSH32}] & \\ i + 1 & \text{otherwise} \end{cases}$$

9.4.4. *Normal Halting.* The normal halting function H is defined:

(142)
$$H(\boldsymbol{\mu}, I) \equiv \begin{cases} H_{\text{RETURN}}(\boldsymbol{\mu}) & \text{if} \quad w \in \{\text{RETURN}, \text{REVERT}\} \\ () & \text{if} \quad w \in \{\text{STOP}, \text{SELFDESTRUCT}\} \\ \varnothing & \text{otherwise} \end{cases}$$

The data-returning halt operations, RETURN and REVERT, have a special function H_{RETURN}. Note also the difference between the empty sequence and the empty set as discussed here.

9.5. **The Execution Cycle.** Stack items are added or removed from the left-most, lower-indexed portion of the series; all other items remain unchanged:

$$O\big((\boldsymbol{\sigma}, \boldsymbol{\mu}, A, I)\big) \equiv (\boldsymbol{\sigma}', \boldsymbol{\mu}', A', I) \tag{143}$$
$$\Delta \equiv \alpha_w - \delta_w \tag{144}$$
$$\|\boldsymbol{\mu}'_{\mathbf{s}}\| \equiv \|\boldsymbol{\mu}_{\mathbf{s}}\| + \Delta \tag{145}$$
$$\forall x \in [\alpha_w, \|\boldsymbol{\mu}'_{\mathbf{s}}\|) : \boldsymbol{\mu}'_{\mathbf{s}}[x] \equiv \boldsymbol{\mu}_{\mathbf{s}}[x - \Delta] \tag{146}$$

The gas is reduced by the instruction's gas cost and for most instructions, the program counter increments on each cycle, for the three exceptions, we assume a function J, subscripted by one of two instructions, which evaluates to the according value:

$$\boldsymbol{\mu}'_g \equiv \boldsymbol{\mu}_g - C(\boldsymbol{\sigma}, \boldsymbol{\mu}, I) \tag{147}$$
$$\boldsymbol{\mu}'_{\text{pc}} \equiv \begin{cases} J_{\text{JUMP}}(\boldsymbol{\mu}) & \text{if} \quad w = \text{JUMP} \\ J_{\text{JUMPI}}(\boldsymbol{\mu}) & \text{if} \quad w = \text{JUMPI} \\ N(\boldsymbol{\mu}_{\text{pc}}, w) & \text{otherwise} \end{cases} \tag{148}$$

In general, we assume the memory, self-destruct set and system state don't change:

$$\boldsymbol{\mu}'_{\mathbf{m}} \equiv \boldsymbol{\mu}_{\mathbf{m}} \tag{149}$$
$$\boldsymbol{\mu}'_{\text{i}} \equiv \boldsymbol{\mu}_{\text{i}} \tag{150}$$
$$A' \equiv A \tag{151}$$
$$\boldsymbol{\sigma}' \equiv \boldsymbol{\sigma} \tag{152}$$

However, instructions do typically alter one or several components of these values. Altered components listed by instruction are noted in Appendix H, alongside values for α and δ and a formal description of the gas requirements.

10. Blocktree to Blockchain

The canonical blockchain is a path from root to leaf through the entire block tree. In order to have consensus over which path it is, conceptually we identify the path that has had the most computation done upon it, or, the *heaviest* path. Clearly one factor that helps determine the heaviest path is the block number of the leaf, equivalent to the number of blocks, not counting the unmined genesis block, in the path. The longer the path, the greater the total mining effort that must have been done in order to arrive at the leaf. This is akin to existing schemes, such as that employed in Bitcoin-derived protocols.

Since a block header includes the difficulty, the header alone is enough to validate the computation done. Any block contributes toward the total computation or *total difficulty* of a chain.

Thus we define the total difficulty of block B recursively as:

$$B_{\text{t}} \equiv B'_{\text{t}} + B_{\text{d}} \tag{153}$$
$$B' \equiv P(B_H) \tag{154}$$

As such given a block B, B_{t} is its total difficulty, B' is its parent block and B_{d} is its difficulty.

11. Block Finalisation

The process of finalising a block involves four stages:

(1) Validate (or, if mining, determine) ommers;
(2) validate (or, if mining, determine) transactions;
(3) apply rewards;
(4) verify (or, if mining, compute a valid) state and block nonce.

11.1. **Ommer Validation.** The validation of ommer headers means nothing more than verifying that each ommer header is both a valid header and satisfies the relation of Nth-generation ommer to the present block where $N \leq 6$. The maximum of ommer headers is two. Formally:

$$\|B_{\mathbf{U}}\| \leqslant 2 \bigwedge_{\mathbf{U} \in B_{\mathbf{U}}} V(\mathbf{U}) \ \wedge \ k(\mathbf{U}, P(\mathbf{B_H})_{\mathbf{H}}, 6) \tag{155}$$

where k denotes the "is-kin" property:

(156)
$$k(U, H, n) \equiv \begin{cases} false & \text{if} \quad n = 0 \\ s(U, H) & \\ \quad \vee \ k(U, P(H)_H, n - 1) & \text{otherwise} \end{cases}$$

and s denotes the "is-sibling" property:

(157)
$$s(U, H) \equiv (P(H) = P(U) \ \wedge \ H \neq U \ \wedge \ U \notin B(H)_{\mathbf{U}})$$

where $B(H)$ and $P(H)$ are the block and the parent block of the corresponding header H respectively.

11.2. **Transaction Validation.** The given **gasUsed** must correspond faithfully to the transactions listed: $B_{H\text{g}}$, the total gas used in the block, must be equal to the accumulated gas used according to the final transaction:

$$B_{H\text{g}} = \ell(\mathbf{R})_{\text{u}} \tag{158}$$

11.3. **Reward Application.** The application of rewards to a block involves raising the balance of the accounts of the beneficiary address of the block and each ommer by a certain amount. We raise the block's beneficiary account by R_{block}; for each ommer, we raise the block's beneficiary by an additional $\frac{1}{32}$ of the block reward and the beneficiary of the ommer gets rewarded depending on the block number. Formally we define the function Ω:

(159)
$$\Omega(B, \boldsymbol{\sigma}) \equiv \boldsymbol{\sigma}' : \boldsymbol{\sigma}' = \boldsymbol{\sigma} \quad \text{except:}$$

(160)
$$\boldsymbol{\sigma}'[\mathbf{B}_{H\text{c}}]_{\text{b}} = \boldsymbol{\sigma}[\mathbf{B}_{H\text{c}}]_{\text{b}} + \left(1 + \frac{\|\mathbf{B}_{\mathbf{U}}\|}{32}\right) R_{\text{block}}$$

(161) $\forall_{\mathbf{U} \in \mathbf{B}_{\mathbf{U}}}$:
$$\boldsymbol{\sigma}'[\mathbf{U}_{\text{c}}] = \begin{cases} \varnothing & \text{if } \boldsymbol{\sigma}[\mathbf{U}_{\text{c}}] = \varnothing \ \wedge \ R = 0 \\ \mathbf{a}' & \text{otherwise} \end{cases}$$

(162)
$$\mathbf{a}' \equiv (\boldsymbol{\sigma}[U_{\text{c}}]_{\text{n}}, \boldsymbol{\sigma}[U_{\text{c}}]_{\text{b}} + R, \boldsymbol{\sigma}[U_{\text{c}}]_{\mathbf{s}}, \boldsymbol{\sigma}[U_{\text{c}}]_{\text{c}})$$

(163)
$$R \equiv \left(1 + \frac{1}{8}(U_{\text{i}} - B_{H\text{i}})\right) R_{\text{block}}$$

If there are collisions of the beneficiary addresses between ommers and the block (i.e. two ommers with the same beneficiary address or an ommer with the same beneficiary address as the present block), additions are applied cumulatively.

We define the block reward as 3 Ether:

(164)
$$\text{Let} \quad R_{\text{block}} = 3 \times 10^{18}$$

11.4. **State & Nonce Validation.** We may now define the function, Γ, that maps a block B to its initiation state:

(165)
$$\Gamma(B) \equiv \begin{cases} \boldsymbol{\sigma}_0 kern10pc \text{ if} \quad P(B_H) = \varnothing \\ \boldsymbol{\sigma}_{\text{i}} : \texttt{TRIE}(L_S(\boldsymbol{\sigma}_{\text{i}})) = P(B_H)_{H_{\text{r}}} \quad \text{otherwise} \end{cases}$$

Here, $\texttt{TRIE}(L_S(\boldsymbol{\sigma}_{\text{i}}))$ means the hash of the root node of a trie of state $\boldsymbol{\sigma}_{\text{i}}$; it is assumed that implementations will store this in the state database, which is trivial and efficient since the trie is by nature an immutable data structure.

And finally we define Φ, the block transition function, which maps an incomplete block B to a complete block B':

(166)
$$\Phi(B) \equiv B' : \quad B' = B^* \quad \text{except:}$$

(167)
$$B'_{\text{n}} = n : \quad x \leqslant \frac{2^{256}}{H_{\text{d}}}$$

(168)
$$B'_{\text{m}} = m \quad \text{with } (x, m) = \texttt{PoW}(B^*_{\not{H}}, n, \mathbf{d})$$

(169)
$$B^* \equiv B \quad \text{except:} \quad B^*_{\text{r}} = r(\Pi(\Gamma(B), B))$$

With $\mathbf{d}$ being a dataset as specified in appendix J.

As specified at the beginning of the present work, Π is the state-transition function, which is defined in terms of Ω, the block finalisation function and Υ, the transaction-evaluation function, both now well-defined.

As previously detailed, $\mathbf{R}[n]_{\text{z}}$, $\mathbf{R}[n]_{\mathbf{l}}$ and $\mathbf{R}[n]_{\text{u}}$ are the nth corresponding status code, logs and cumulative gas used after each transaction ($\mathbf{R}[n]_{\text{b}}$, the fourth component in the tuple, has already been defined in terms of the logs). We also define the nth state $\boldsymbol{\sigma}[n]$, which is defined simply as the state resulting from applying the corresponding transaction to the state resulting from the previous transaction (or the block's initial state in the case of the first such transaction):

(170)
$$\boldsymbol{\sigma}[n] = \begin{cases} \Gamma(B) & \text{if} \quad n < 0 \\ \Upsilon(\boldsymbol{\sigma}[n-1], B_{\mathbf{T}}[n]) & \text{otherwise} \end{cases}$$

In the case of $B_{\mathbf{R}}[n]_{\text{u}}$, we take a similar approach defining each item as the gas used in evaluating the corresponding transaction summed with the previous item (or zero, if it is the first), giving us a running total:

(171)
$$\mathbf{R}[n]_{\text{u}} = \begin{cases} 0 & \text{if} \quad n < 0 \\ \Upsilon^g(\boldsymbol{\sigma}[n-1], B_{\mathbf{T}}[n]) & \\ \quad +\mathbf{R}[n-1]_{\text{u}} & \text{otherwise} \end{cases}$$

For $\mathbf{R}[n]_{\mathbf{l}}$, we utilise the $\Upsilon^{\mathbf{l}}$ function that we conveniently defined in the transaction execution function.

(172)
$$\mathbf{R}[n]_{\mathbf{l}} = \Upsilon^{\mathbf{l}}(\boldsymbol{\sigma}[n-1], B_{\mathbf{T}}[n])$$

We define $\mathbf{R}[n]_{\text{z}}$ in a similar manner.

(173)
$$\mathbf{R}[n]_{\text{z}} = \Upsilon^{\text{z}}(\boldsymbol{\sigma}[n-1], B_{\mathbf{T}}[n])$$

Finally, we define Π as the new state given the block reward function Ω applied to the final transaction's resultant state, $\ell(\boldsymbol{\sigma})$:

(174)
$$\Pi(\boldsymbol{\sigma}, B) \equiv \Omega(B, \ell(\boldsymbol{\sigma}))$$

Thus the complete block-transition mechanism is defined, except for $\texttt{PoW}$, the proof-of-work function.

11.5. **Mining Proof-of-Work.** The mining proof-of-work (PoW) exists as a cryptographically secure nonce that proves beyond reasonable doubt that a particular amount of computation has been expended in the determination of some token value n. It is utilised to enforce the blockchain security by giving meaning and credence to the notion of difficulty (and, by extension, total difficulty). However, since mining new blocks comes with an attached reward, the proof-of-work not only functions as a method of securing confidence that the blockchain will remain canonical into the future, but also as a wealth distribution mechanism.

For both reasons, there are two important goals of the proof-of-work function; firstly, it should be as accessible as possible to as many people as possible. The requirement of, or reward from, specialised and uncommon hardware should be minimised. This makes the distribution model as open as possible, and, ideally, makes the act of mining a simple swap from electricity to Ether at roughly the same rate for anyone around the world.

Secondly, it should not be possible to make super-linear profits, and especially not so with a high initial barrier. Such a mechanism allows a well-funded adversary to gain a troublesome amount of the network's total mining power and as such gives them a super-linear reward (thus skewing distribution in their favour) as well as reducing the network security.

One plague of the Bitcoin world is ASICs. These are specialised pieces of compute hardware that exist only to do a single task (Smith [1997]). In Bitcoin's case the task is the SHA256 hash function (Courtois et al. [2014]). While ASICs exist for a proof-of-work function, both goals are placed in jeopardy. Because of this, a proof-of-work function that is ASIC-resistant (i.e. difficult or economically inefficient to implement in specialised compute hardware) has been identified as the proverbial silver bullet.

Two directions exist for ASIC resistance; firstly make it sequential memory-hard, i.e. engineer the function such that the determination of the nonce requires a lot of memory and bandwidth such that the memory cannot be used in parallel to discover multiple nonces simultaneously. The second is to make the type of computation it would need to do general-purpose; the meaning of "specialised hardware" for a general-purpose task set is, naturally, general purpose hardware and as such commodity desktop computers are likely to be pretty close to "specialised hardware" for the task. For Ethereum 1.0 we have chosen the first path.

More formally, the proof-of-work function takes the form of PoW:

(175)

$$m = H_{\mathrm{m}} \quad \wedge \quad n \leqslant \frac{2^{256}}{H_{\mathrm{d}}} \quad \text{with} \quad (m, n) = \mathtt{PoW}(H_{\mathbb{R}}, H_{\mathrm{n}}, \mathbf{d})$$

Where $H_{\mathbb{R}}$ is the new block's header but *without* the nonce and mix-hash components; H_{n} is the nonce of the header; $\mathbf{d}$ is a large data set needed to compute the mixHash and H_{d} is the new block's difficulty value (i.e. the block difficulty from section 10). PoW is the proof-of-work function which evaluates to an array with the first item being the mixHash and the second item being a pseudo-random number cryptographically dependent on H and $\mathbf{d}$. The underlying algorithm is called Ethash and is described below.

11.5.1. *Ethash.* Ethash is the PoW algorithm for Ethereum 1.0. It is the latest version of Dagger-Hashimoto, introduced by Buterin [2013b] and Dryja [2014], although it can no longer appropriately be called that since many of the original features of both algorithms were drastically changed with R&D from February 2015 until May 4 2015 (Jentzsch [2015]). The general route that the algorithm takes is as follows:

There exists a seed which can be computed for each block by scanning through the block headers up until that point. From the seed, one can compute a pseudorandom cache, $J_{cacheinit}$ bytes in initial size. Light clients store the cache. From the cache, we can generate a dataset, $J_{datasetinit}$ bytes in initial size, with the property that each item in the dataset depends on only a small number of items from the cache. Full clients and miners store the dataset. The dataset grows linearly with time.

Mining involves grabbing random slices of the dataset and hashing them together. Verification can be done with low memory by using the cache to regenerate the specific pieces of the dataset that you need, so you only need to store the cache. The large dataset is updated once every J_{epoch} blocks, so the vast majority of a miner's effort will be reading the dataset, not making changes to it. The mentioned parameters as well as the algorithm is explained in detail in appendix J.

12. Implementing Contracts

There are several patterns of contracts engineering that allow particular useful behaviours; two of these that we will briefly discuss are data feeds and random numbers.

12.1. **Data Feeds.** A data feed contract is one which provides a single service: it gives access to information from the external world within Ethereum. The accuracy and timeliness of this information is not guaranteed and it is the task of a secondary contract author—the contract that utilises the data feed—to determine how much trust can be placed in any single data feed.

The general pattern involves a single contract within Ethereum which, when given a message call, replies with some timely information concerning an external phenomenon. An example might be the local temperature of New York City. This would be implemented as a contract that returned that value of some known point in storage. Of course this point in storage must be maintained with the correct such temperature, and thus the second part of the pattern would be for an external server to run an Ethereum node, and immediately on discovery of a new block, creates a new valid transaction, sent to the contract, updating said value in storage. The contract's code would accept such updates only from the identity contained on said server.

12.2. **Random Numbers.** Providing random numbers within a deterministic system is, naturally, an impossible task. However, we can approximate with pseudo-random numbers by utilising data which is generally unknowable at the time of transacting. Such data might include the block's hash, the block's timestamp and the block's beneficiary address. In order to make it hard for malicious miners to control those values, one should use the BLOCKHASH operation in order to use hashes of the previous 256 blocks as pseudo-random numbers. For a series of such numbers, a trivial solution would be to add some constant amount and hashing the result.

13. Future Directions

The state database won't be forced to maintain all past state trie structures into the future. It should maintain an age for each node and eventually discard nodes that are neither recent enough nor checkpoints. Checkpoints, or a set of nodes in the database that allow a particular block's state trie to be traversed, could be used to place a maximum limit on the amount of computation needed in order to retrieve any state throughout the blockchain.

Blockchain consolidation could be used in order to reduce the amount of blocks a client would need to download to act as a full, mining, node. A compressed archive of the trie structure at given points in time (perhaps one in every 10,000th block) could be maintained by the peer network, effectively recasting the genesis block. This would reduce the amount to be downloaded to a single archive plus a hard maximum limit of blocks.

Finally, blockchain compression could perhaps be conducted: nodes in state trie that haven't sent/received a transaction in some constant amount of blocks could be thrown out, reducing both Ether-leakage and the growth of the state database.

13.1. **Scalability.** Scalability remains an eternal concern. With a generalised state transition function, it becomes difficult to partition and parallelise transactions to apply the divide-and-conquer strategy. Unaddressed, the dynamic value-range of the system remains essentially fixed and as the average transaction value increases, the less valuable of them become ignored, being economically pointless to include in the main ledger. However, several strategies exist that may potentially be exploited to provide a considerably more scalable protocol.

Some form of hierarchical structure, achieved by either consolidating smaller lighter-weight chains into the main

block or building the main block through the incremental combination and adhesion (through proof-of-work) of smaller transaction sets may allow parallelisation of transaction combination and block-building. Parallelism could also come from a prioritised set of parallel blockchains, consolidating each block and with duplicate or invalid transactions thrown out accordingly.

Finally, verifiable computation, if made generally available and efficient enough, may provide a route to allow the proof-of-work to be the verification of final state.

14. Conclusion

We have introduced, discussed and formally defined the protocol of Ethereum. Through this protocol the reader may implement a node on the Ethereum network and join others in a decentralised secure social operating system. Contracts may be authored in order to algorithmically specify and autonomously enforce rules of interaction.

15. Acknowledgements

Many thanks to Aeron Buchanan for authoring the *Homestead* revisions, Christoph Jentzsch for authoring the Ethash algorithm and Yoichi Hirai for doing most of the EIP-150 changes. Important maintenance, useful corrections and suggestions were provided by a number of others from the Ethereum DEV organisation and Ethereum community at large including Gustav Simonsson, Paweł Bylica, Jutta Steiner, Nick Savers, Viktor Trón, Marko Simovic, Giacomo Tazzari and, of course, Vitalik Buterin.

16. Availability

The source of this paper is maintained at `https://github.com/ethereum/yellowpaper/`. An auto-generated PDF is located at `https://ethereum.github.io/yellowpaper/paper.pdf`.

References

Pierre Arnaud, Mathieu Schroeter, and Sam Le Barbare. Electrum, 2017. URL `https://www.npmjs.com/package/electrum`.

Jacob Aron. BitCoin software finds new life. *New Scientist*, 213(2847):20, 2012. URL `http://www.sciencedirect.com/science/article/pii/S0262407912601055`.

Adam Back. Hashcash - Amortizable Publicly Auditable Cost-Functions, 2002. URL `http://www.hashcash.org/papers/amortizable.pdf`.

Guido Bertoni, Joan Daemen, Michal Peeters, Gilles Van Assche, and Ronny Van Keer. KECCAK, 2017. URL `https://keccak.team/keccak.html`.

Roman Boutellier and Mareike Heinzen. Pirates, Pioneers, Innovators and Imitators. In *Growth Through Innovation*, pages 85–96. Springer, 2014. URL `https://www.springer.com/gb/book/9783319040158`.

Vitalik Buterin. Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform, 2013a. URL `https://github.com/ethereum/wiki/wiki/White-Paper`.

Vitalik Buterin. Dagger: A Memory-Hard to Compute, Memory-Easy to Verify Scrypt Alternative, 2013b. URL `http://www.hashcash.org/papers/dagger.html`.

Vitalik Buterin. EIP-2: Homestead hard-fork changes, 2015. URL `https://github.com/ethereum/EIPs/blob/master/EIPS/eip-2.md`.

Vitalik Buterin. EIP-100: Change difficulty adjustment to target mean block time including uncles, April 2016. URL `https://github.com/ethereum/EIPs/blob/master/EIPS/eip-100.md`.

Nicolas T. Courtois, Marek Grajek, and Rahul Naik. *Optimizing SHA256 in Bitcoin Mining*, pages 131–144. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014. ISBN 978-3-662-44893-9. doi: 10.1007/978-3-662-44893-9_12. URL `https://doi.org/10.1007/978-3-662-44893-9_12`.

B.A. Davey and H.A. Priestley. *Introduction to lattices and order. 2nd ed.* Cambridge: Cambridge University Press, 2nd ed. edition, 2002. ISBN 0-521-78451-4/pbk.

Thaddeus Dryja. Hashimoto: I/O bound proof of work, 2014. URL `http://diyhpl.us/~bryan/papers2/bitcoin/meh/hashimoto.pdf`.

Cynthia Dwork and Moni Naor. Pricing via processing or combatting junk mail. In *In 12th Annual International Cryptology Conference*, pages 139–147, 1992. URL `http://www.wisdom.weizmann.ac.il/~naor/PAPERS/pvp.pdf`.

Phong Vo Glenn Fowler, Landon Curt Noll. FowlerNollVo hash function, 1991. URL `http://www.isthe.com/chongo/tech/comp/fnv/index.html`.

Nils Gura, Arun Patel, Arvinderpal Wander, Hans Eberle, and Sheueling Chang Shantz. Comparing elliptic curve cryptography and RSA on 8-bit CPUs. In *Cryptographic Hardware and Embedded Systems-CHES 2004*, pages 119–132. Springer, 2004. URL `https://www.iacr.org/archive/ches2004/31560117/31560117.pdf`.

Christoph Jentzsch. Commit date for ethash, 2015. URL `https://github.com/ethereum/yellowpaper/commit/77a8cf2428ce245bf6e2c39c5e652ba58a278666#commitcomment-24644869`.

Don Johnson, Alfred Menezes, and Scott Vanstone. The Elliptic Curve Digital Signature Algorithm (ECDSA), 2001. URL `https://web.archive.org/web/20170921160141/http://cs.ucsb.edu/~koc/ccs130h/notes/ecdsa-cert.pdf`. Accessed 21 September 2017, but the original link was inaccessible on 19 October 2017. Refer to section 6.2 for ECDSAPUBKEY, and section 7 for ECDSASIGN and ECDSARECOVER.

Sergio Demian Lerner. Strict Memory Hard Hashing Functions, 2014. URL `http://www.hashcash.org/papers/memohash.pdf`.

Mark Miller. The Future of Law. In *paper delivered at the Extro 3 Conference (August 9)*, 1997. URL `https://drive.google.com/file/d/0Bw0VXJKBgYPMS0J2VGIyWWlocms/edit?usp=sharing`.

Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008. URL `http://www.bitcoin.org/bitcoin.pdf`.

Meni Rosenfeld, Yoni Assia, Vitalik Buterin, m liorhakiLior, Oded Leiba, Assaf Shomer, and Eliran Zach. Colored Coins Protocol Specification, 2012. URL `https://github.com/Colored-Coins/Colored-Coins-Protocol-Specification`.

Afri Schoedon and Vitalik Buterin. EIP-649: Metropolis difficulty bomb delay and block reward reduction,

June 2017. URL https://github.com/ethereum/EIPs/blob/master/EIPS/eip-649.md.

Michael John Sebastian Smith. *Application-Specific Integrated Circuits*. Addison-Wesley, 1997. ISBN 0201500221.

Yonatan Sompolinsky and Aviv Zohar. Accelerating bitcoin's transaction processing. fast money grows on trees, not chains, 2013. URL https://eprint.iacr.org/2013/881.

Simon Sprankel. Technical Basis of Digital Currencies, 2013. URL http://www.coderblog.de/wp-content/uploads/technical-basis-of-digital-currencies.pdf.

Nick Szabo. Formalizing and securing relationships on public networks. *First Monday*, 2(9), 1997. URL http://firstmonday.org/ojs/index.php/fm/article/view/548.

Vivek Vishnumurthy, Sangeeth Chandrakumar, and Emin Gn Sirer. KARMA: A secure economic framework for peer-to-peer resource sharing, 2003. URL https://www.cs.cornell.edu/people/egs/papers/karma.pdf.

J. R. Willett. MasterCoin Complete Specification, 2013. URL https://github.com/mastercoin-MSC/spec.

Appendix A. Terminology

External Actor: A person or other entity able to interface to an Ethereum node, but external to the world of Ethereum. It can interact with Ethereum through depositing signed Transactions and inspecting the blockchain and associated state. Has one (or more) intrinsic Accounts.

Address: A 160-bit code used for identifying Accounts.

Account: Accounts have an intrinsic balance and transaction count maintained as part of the Ethereum state. They also have some (possibly empty) EVM Code and a (possibly empty) Storage State associated with them. Though homogenous, it makes sense to distinguish between two practical types of account: those with empty associated EVM Code (thus the account balance is controlled, if at all, by some external entity) and those with non-empty associated EVM Code (thus the account represents an Autonomous Object). Each Account has a single Address that identifies it.

Transaction: A piece of data, signed by an External Actor. It represents either a Message or a new Autonomous Object. Transactions are recorded into each block of the blockchain.

Autonomous Object: A notional object existent only within the hypothetical state of Ethereum. Has an intrinsic address and thus an associated account; the account will have non-empty associated EVM Code. Incorporated only as the Storage State of that account.

Storage State: The information particular to a given Account that is maintained between the times that the Account's associated EVM Code runs.

Message: Data (as a set of bytes) and Value (specified as Ether) that is passed between two Accounts, either through the deterministic operation of an Autonomous Object or the cryptographically secure signature of the Transaction.

Message Call: The act of passing a message from one Account to another. If the destination account is associated with non-empty EVM Code, then the VM will be started with the state of said Object and the Message acted upon. If the message sender is an Autonomous Object, then the Call passes any data returned from the VM operation.

Gas: The fundamental network cost unit. Paid for exclusively by Ether (as of PoC-4), which is converted freely to and from Gas as required. Gas does not exist outside of the internal Ethereum computation engine; its price is set by the Transaction and miners are free to ignore Transactions whose Gas price is too low.

Contract: Informal term used to mean both a piece of EVM Code that may be associated with an Account or an Autonomous Object.

Object: Synonym for Autonomous Object.

App: An end-user-visible application hosted in the Ethereum Browser.

Ethereum Browser: (aka Ethereum Reference Client) A cross-platform GUI of an interface similar to a simplified browser (a la Chrome) that is able to host sandboxed applications whose backend is purely on the Ethereum protocol.

Ethereum Virtual Machine: (aka EVM) The virtual machine that forms the key part of the execution model for an Account's associated EVM Code.

Ethereum Runtime Environment: (aka ERE) The environment which is provided to an Autonomous Object executing in the EVM. Includes the EVM but also the structure of the world state on which the EVM relies for certain I/O instructions including CALL & CREATE.

EVM Code: The bytecode that the EVM can natively execute. Used to formally specify the meaning and ramifications of a message to an Account.

EVM Assembly: The human-readable form of EVM-code.

LLL: The Lisp-like Low-level Language, a human-writable language used for authoring simple contracts and general low-level language toolkit for trans-compiling to.

Appendix B. Recursive Length Prefix

This is a serialisation method for encoding arbitrarily structured binary data (byte arrays).

We define the set of possible structures $\mathbb{T}$:

$$\mathbb{T} \equiv \mathbb{L} \cup \mathbb{B} \tag{176}$$

$$\mathbb{L} \equiv \{\mathbf{t} : \mathbf{t} = (\mathbf{t}[0], \mathbf{t}[1], ...) \ \wedge \ \forall_{n<\|\mathbf{t}\|} \ \mathbf{t}[n] \in \mathbb{T}\} \tag{177}$$

$$\mathbb{B} \equiv \{\mathbf{b} : \mathbf{b} = (\mathbf{b}[0], \mathbf{b}[1], ...) \ \wedge \ \forall_{n<\|\mathbf{b}\|} \ \mathbf{b}[n] \in \mathbb{O}\} \tag{178}$$

Where $\mathbb{O}$ is the set of (8-bit) bytes. Thus $\mathbb{B}$ is the set of all sequences of bytes (otherwise known as byte-arrays, and a leaf if imagined as a tree), $\mathbb{L}$ is the set of all tree-like (sub-)structures that are not a single leaf (a branch node if imagined as a tree) and $\mathbb{T}$ is the set of all byte-arrays and such structural sequences.

We define the RLP function as $\mathtt{RLP}$ through two sub-functions, the first handling the instance when the value is a byte array, the second when it is a sequence of further values:

$$\mathtt{RLP}(\mathbf{x}) \equiv \begin{cases} R_{\mathrm{b}}(\mathbf{x}) & \text{if} \quad \mathbf{x} \in \mathbb{B} \\ R_{\mathrm{l}}(\mathbf{x}) & \text{otherwise} \end{cases} \tag{179}$$

If the value to be serialised is a byte-array, the RLP serialisation takes one of three forms:

- If the byte-array contains solely a single byte and that single byte is less than 128, then the input is exactly equal to the output.
- If the byte-array contains fewer than 56 bytes, then the output is equal to the input prefixed by the byte equal to the length of the byte array plus 128.
- Otherwise, the output is equal to the input prefixed by the minimal-length byte-array which when interpreted as a big-endian integer is equal to the length of the input byte array, which is itself prefixed by the number of bytes required to faithfully encode this length value plus 183.

Formally, we define R_{b}:

$$R_{\mathrm{b}}(\mathbf{x}) \equiv \begin{cases} \mathbf{x} & \text{if} \quad \|\mathbf{x}\| = 1 \wedge \mathbf{x}[0] < 128 \\ (128 + \|\mathbf{x}\|) \cdot \mathbf{x} & \text{else if} \quad \|\mathbf{x}\| < 56 \\ \left(183 + \left\|\mathtt{BE}(\|\mathbf{x}\|)\right\|\right) \cdot \mathtt{BE}(\|\mathbf{x}\|) \cdot \mathbf{x} & \text{otherwise} \end{cases} \tag{180}$$

$$\mathtt{BE}(x) \equiv (b_0, b_1, ...) : b_0 \neq 0 \wedge x = \sum_{n=0}^{n<\|\mathbf{b}\|} b_{\mathrm{n}} \cdot 256^{\|\mathbf{b}\|-1-n} \tag{181}$$

$$(a) \cdot (b, c) \cdot (d, e) = (a, b, c, d, e) \tag{182}$$

Thus $\mathtt{BE}$ is the function that expands a non-negative integer value to a big-endian byte array of minimal length and the dot operator performs sequence concatenation.

If instead, the value to be serialised is a sequence of other items then the RLP serialisation takes one of two forms:

- If the concatenated serialisations of each contained item is less than 56 bytes in length, then the output is equal to that concatenation prefixed by the byte equal to the length of this byte array plus 192.
- Otherwise, the output is equal to the concatenated serialisations prefixed by the minimal-length byte-array which when interpreted as a big-endian integer is equal to the length of the concatenated serialisations byte array, which is itself prefixed by the number of bytes required to faithfully encode this length value plus 247.

Thus we finish by formally defining R_{l}:

$$R_{\mathrm{l}}(\mathbf{x}) \equiv \begin{cases} (192 + \|s(\mathbf{x})\|) \cdot s(\mathbf{x}) & \text{if} \quad \|s(\mathbf{x})\| < 56 \\ \left(247 + \left\|\mathtt{BE}(\|s(\mathbf{x})\|)\right\|\right) \cdot \mathtt{BE}(\|s(\mathbf{x})\|) \cdot s(\mathbf{x}) & \text{otherwise} \end{cases} \tag{183}$$

$$s(\mathbf{x}) \equiv \mathtt{RLP}(\mathbf{x}_0) \cdot \mathtt{RLP}(\mathbf{x}_1)... \tag{184}$$

If RLP is used to encode a scalar, defined only as a non-negative integer ($\mathbb{N}$ or any x for $\mathbb{N}_{\mathrm{x}}$), it must be specified as the shortest byte array such that the big-endian interpretation of it is equal. Thus the RLP of some non-negative integer i is defined as:

$$\mathtt{RLP}(i : i \in \mathbb{N}) \equiv \mathtt{RLP}(\mathtt{BE}(i)) \tag{185}$$

When interpreting RLP data, if an expected fragment is decoded as a scalar and leading zeroes are found in the byte sequence, clients are required to consider it non-canonical and treat it in the same manner as otherwise invalid RLP data, dismissing it completely.

There is no specific canonical encoding format for signed or floating-point values.

Appendix C. Hex-Prefix Encoding

Hex-prefix encoding is an efficient method of encoding an arbitrary number of nibbles as a byte array. It is able to store an additional flag which, when used in the context of the trie (the only context in which it is used), disambiguates between node types.

It is defined as the function $\mathtt{HP}$ which maps from a sequence of nibbles (represented by the set $\mathbb{Y}$) together with a boolean value to a sequence of bytes (represented by the set $\mathbb{B}$):

$$\mathtt{HP}(\mathbf{x}, t) : \mathbf{x} \in \mathbb{Y} \equiv \begin{cases} (16f(t), 16\mathbf{x}[0] + \mathbf{x}[1], 16\mathbf{x}[2] + \mathbf{x}[3], ...) & \text{if} \quad \|\mathbf{x}\| \text{ is even} \\ (16(f(t)+1) + \mathbf{x}[0], 16\mathbf{x}[1] + \mathbf{x}[2], 16\mathbf{x}[3] + \mathbf{x}[4], ...) & \text{otherwise} \end{cases} \tag{186}$$

$$f(t) \equiv \begin{cases} 2 & \text{if} \quad t \neq 0 \\ 0 & \text{otherwise} \end{cases} \tag{187}$$

Thus the high nibble of the first byte contains two flags; the lowest bit encoding the oddness of the length and the second-lowest encoding the flag t. The low nibble of the first byte is zero in the case of an even number of nibbles and the first nibble in the case of an odd number. All remaining nibbles (now an even number) fit properly into the remaining bytes.

Appendix D. Modified Merkle Patricia Tree

The modified Merkle Patricia tree (trie) provides a persistent data structure to map between arbitrary-length binary data (byte arrays). It is defined in terms of a mutable data structure to map between 256-bit binary fragments and arbitrary-length binary data, typically implemented as a database. The core of the trie, and its sole requirement in terms of the protocol specification is to provide a single value that identifies a given set of key-value pairs, which may be either a 32-byte sequence or the empty byte sequence. It is left as an implementation consideration to store and maintain the structure of the trie in a manner that allows effective and efficient realisation of the protocol.

Formally, we assume the input value $\mathfrak{I}$, a set containing pairs of byte sequences:

$$\mathfrak{I} = \{(\mathbf{k}_0 \in \mathbb{B}, \mathbf{v}_0 \in \mathbb{B}), (\mathbf{k}_1 \in \mathbb{B}, \mathbf{v}_1 \in \mathbb{B}), ...\} \tag{188}$$

When considering such a sequence, we use the common numeric subscript notation to refer to a tuple's key or value, thus:

$$\forall_{\mathbf{I} \in \mathfrak{I}} I \equiv (I_0, I_1) \tag{189}$$

Any series of bytes may also trivially be viewed as a series of nibbles, given an endian-specific notation; here we assume big-endian. Thus:

$$y(\mathfrak{I}) = \{(\mathbf{k}'_0 \in \mathbb{Y}, \mathbf{v}_0 \in \mathbb{B}), (\mathbf{k}'_1 \in \mathbb{Y}, \mathbf{v}_1 \in \mathbb{B}), ...\} \tag{190}$$

$$\forall_n \quad \forall_{i : i < 2\|\mathbf{k}_n\|} \quad \mathbf{k}'_n[i] \equiv \begin{cases} \lfloor \mathbf{k}_n[i \div 2] \div 16 \rfloor & \text{if } i \text{ is even} \\ \mathbf{k}_n[\lfloor i \div 2 \rfloor] \bmod 16 & \text{otherwise} \end{cases} \tag{191}$$

We define the function TRIE, which evaluates to the root of the trie that represents this set when encoded in this structure:

$$\mathtt{TRIE}(\mathfrak{I}) \equiv \mathtt{KEC}(c(\mathfrak{I}, 0)) \tag{192}$$

We also assume a function n, the trie's node cap function. When composing a node, we use RLP to encode the structure. As a means of reducing storage complexity, for nodes whose composed RLP is fewer than 32 bytes, we store the RLP directly; for those larger we assert prescience of the byte array whose Keccak hash evaluates to our reference. Thus we define in terms of c, the node composition function:

$$n(\mathfrak{I}, i) \equiv \begin{cases} () & \text{if} \quad \mathfrak{I} = \varnothing \\ c(\mathfrak{I}, i) & \text{if} \quad \|c(\mathfrak{I}, i)\| < 32 \\ \mathtt{KEC}(c(\mathfrak{I}, i)) & \text{otherwise} \end{cases} \tag{193}$$

In a manner similar to a radix tree, when the trie is traversed from root to leaf, one may build a single key-value pair. The key is accumulated through the traversal, acquiring a single nibble from each branch node (just as with a radix tree). Unlike a radix tree, in the case of multiple keys sharing the same prefix or in the case of a single key having a unique suffix, two optimising nodes are provided. Thus while traversing, one may potentially acquire multiple nibbles from each of the other two node types, extension and leaf. There are three kinds of nodes in the trie:

Leaf: A two-item structure whose first item corresponds to the nibbles in the key not already accounted for by the accumulation of keys and branches traversed from the root. The hex-prefix encoding method is used and the second parameter to the function is required to be *true*.

Extension: A two-item structure whose first item corresponds to a series of nibbles of size greater than one that are shared by at least two distinct keys past the accumulation of the keys of nibbles and the keys of branches as traversed from the root. The hex-prefix encoding method is used and the second parameter to the function is required to be *false*.

Branch: A 17-item structure whose first sixteen items correspond to each of the sixteen possible nibble values for the keys at this point in their traversal. The 17th item is used in the case of this being a terminator node and thus a key being ended at this point in its traversal.

A branch is then only used when necessary; no branch nodes may exist that contain only a single non-zero entry. We may formally define this structure with the structural composition function c:

(194)
$$c(\mathfrak{I}, i) \equiv \begin{cases} \mathtt{RLP}\Big(\big(\mathtt{HP}(I_0[i..(\|I_0\|-1)], \mathit{true}), I_1\big)\Big) & \text{if} \quad \|\mathfrak{I}\| = 1 \quad \text{where } \exists I : I \in \mathfrak{I} \\ \mathtt{RLP}\Big(\big(\mathtt{HP}(I_0[i..(j-1)], \mathit{false}), n(\mathfrak{I}, j)\big)\Big) & \text{if} \quad i \neq j \quad \text{where } j = \arg\max_x : \exists \mathbf{l} : \|\mathbf{l}\| = x : \forall_{I \in \mathfrak{I}} : I_0[0..(x-1)] = \mathbf{l} \\ \mathtt{RLP}\Big((u(0), u(1), ..., u(15), v)\Big) & \text{otherwise} \quad \text{where } u(j) \equiv n(\{I : I \in \mathfrak{I} \wedge I_0[i] = j\}, i+1) \\ & \qquad\qquad\qquad\qquad\qquad\qquad v = \begin{cases} I_1 & \text{if} \quad \exists I : I \in \mathfrak{I} \wedge \|I_0\| = i \\ () & \text{otherwise} \end{cases} \end{cases}$$

D.1. **Trie Database.** Thus no explicit assumptions are made concerning what data is stored and what is not, since that is an implementation-specific consideration; we simply define the identity function mapping the key-value set $\mathfrak{I}$ to a 32-byte hash and assert that only a single such hash exists for any $\mathfrak{I}$, which though not strictly true is accurate within acceptable precision given the Keccak hash's collision resistance. In reality, a sensible implementation will not fully recompute the trie root hash for each set.

A reasonable implementation will maintain a database of nodes determined from the computation of various tries or, more formally, it will memoise the function c. This strategy uses the nature of the trie to both easily recall the contents of any previous key-value set and to store multiple such sets in a very efficient manner. Due to the dependency relationship, Merkle-proofs may be constructed with an $O(\log N)$ space requirement that can demonstrate a particular leaf must exist within a trie of a given root hash.

Appendix E. Precompiled Contracts

For each precompiled contract, we make use of a template function, $\Xi_{\mathtt{PRE}}$, which implements the out-of-gas checking.

(195)
$$\Xi_{\mathtt{PRE}}(\boldsymbol{\sigma}, g, I, T) \equiv \begin{cases} (\varnothing, 0, A^0, ()) & \text{if} \quad g < g_{\mathrm{r}} \\ (\boldsymbol{\sigma}, g - g_{\mathrm{r}}, A^0, \mathbf{o}) & \text{otherwise} \end{cases}$$

The precompiled contracts each use these definitions and provide specifications for the $\mathbf{o}$ (the output data) and g_{r}, the gas requirements.

We define $\Xi_{\mathtt{ECREC}}$ as a precompiled contract for the elliptic curve digital signature algorithm (ECDSA) public key recovery function (ecrecover). See Appendix F for the definition of the function $\mathtt{ECDSARECOVER}$. We also define $\mathbf{d}$ to be the input data, well-defined for an infinite length by appending zeroes as required. In the case of an invalid signature ($\mathtt{ECDSARECOVER}(h, v, r, s) = \varnothing$), we return no output.

(196) $$\Xi_{\mathtt{ECREC}} \equiv \Xi_{\mathtt{PRE}} \quad \text{where:}$$

(197) $$g_{\mathrm{r}} = 3000$$

(198) $$|\mathbf{o}| = \begin{cases} 0 & \text{if} \quad \mathtt{ECDSARECOVER}(h, v, r, s) = \varnothing \\ 32 & \text{otherwise} \end{cases}$$

(199) $$\text{if} \quad |\mathbf{o}| = 32 :$$

(200) $$\mathbf{o}[0..11] = 0$$

(201) $$\mathbf{o}[12..31] = \mathtt{KEC}\big(\mathtt{ECDSARECOVER}(h, v, r, s)\big)[12..31] \quad \text{where:}$$

(202) $$\mathbf{d}[0..(|I_{\mathbf{d}}| - 1)] = I_{\mathbf{d}}$$

(203) $$\mathbf{d}[|I_{\mathbf{d}}|..] = (0, 0, ...)$$

(204) $$h = \mathbf{d}[0..31]$$

(205) $$v = \mathbf{d}[32..63]$$

(206) $$r = \mathbf{d}[64..95]$$

(207) $$s = \mathbf{d}[96..127]$$

We define $\Xi_{\mathtt{SHA256}}$ and $\Xi_{\mathtt{RIP160}}$ as precompiled contracts implementing the SHA2-256 and RIPEMD-160 hash functions respectively. Their gas usage is dependent on the input data size, a factor rounded up to the nearest number of words.

$$\Xi_{\texttt{SHA256}} \equiv \Xi_{\texttt{PRE}} \quad \text{where:} \tag{208}$$
$$g_{\text{r}} = 60 + 12\left\lceil \frac{|I_{\mathbf{d}}|}{32} \right\rceil \tag{209}$$
$$\mathbf{o}[0..31] = \texttt{SHA256}(I_{\mathbf{d}}) \tag{210}$$
$$\Xi_{\texttt{RIP160}} \equiv \Xi_{\texttt{PRE}} \quad \text{where:} \tag{211}$$
$$g_{\text{r}} = 600 + 120\left\lceil \frac{|I_{\mathbf{d}}|}{32} \right\rceil \tag{212}$$
$$\mathbf{o}[0..11] = 0 \tag{213}$$
$$\mathbf{o}[12..31] = \texttt{RIPEMD160}(I_{\mathbf{d}}) \tag{214}$$

For the purposes here, we assume we have well-defined standard cryptographic functions for RIPEMD-160 and SHA2-256 of the form:

$$\texttt{SHA256}(\mathbf{i} \in \mathbb{B}) \equiv o \in \mathbb{B}_{32} \tag{215}$$
$$\texttt{RIPEMD160}(\mathbf{i} \in \mathbb{B}) \equiv o \in \mathbb{B}_{20} \tag{216}$$

The fourth contract, the identity function $\Xi_{\texttt{ID}}$ simply defines the output as the input:

$$\Xi_{\texttt{ID}} \equiv \Xi_{\texttt{PRE}} \quad \text{where:} \tag{217}$$
$$g_{\text{r}} = 15 + 3\left\lceil \frac{|I_{\mathbf{d}}|}{32} \right\rceil \tag{218}$$
$$\mathbf{o} = I_{\mathbf{d}} \tag{219}$$

The fifth contract performs arbitrary-precision exponentiation under modulo. Here, 0^0 is taken to be one, and $x \bmod 0$ is zero for all x. The first word in the input specifies the number of bytes that the first non-negative integer B occupies. The second word in the input specifies the number of bytes that the second non-negative integer E occupies. The third word in the input specifies the number of bytes that the third non-negative integer M occupies. These three words are followed by B, E and M. The rest of the input is discarded. Whenever the input is too short, the missing bytes are considered to be zero. The output is encoded big-endian into the same format as M's.

$$\Xi_{\texttt{EXPMOD}} \equiv \Xi_{\texttt{PRE}} \quad \text{except:} \tag{220}$$
$$g_{\text{r}} = \left\lfloor \frac{f\big(\max(\ell_M, \ell_B)\big)\max(\ell'_E, 1)}{G_{quaddivisor}} \right\rfloor \tag{221}$$
$$f(x) \equiv \begin{cases} x^2 & \text{if } x \leq 64 \\ \left\lfloor \frac{x^2}{4} \right\rfloor + 96x - 3072 & \text{if } 64 < x \leq 1024 \\ \left\lfloor \frac{x^2}{16} \right\rfloor + 480x - 199680 & \text{otherwise} \end{cases} \tag{222}$$
$$\ell'_E = \begin{cases} 0 & \text{if } \ell_E \leq 32 \wedge E = 0 \\ \lfloor \log_2(E) \rfloor & \text{if } \ell_E \leq 32 \wedge E \neq 0 \\ 8(\ell_E - 32) + \lfloor \log_2(i[(96+\ell_B)..(127+\ell_B)]) \rfloor & \text{if } 32 < \ell_E \wedge i[(96+\ell_B)..(127+\ell_B)] \neq 0 \\ 8(\ell_E - 32) & \text{otherwise} \end{cases} \tag{223}$$
$$\mathbf{o} = \left(B^E \bmod M\right) \in \mathbb{N}_{8\ell_M} \tag{224}$$
$$\ell_B \equiv i[0..31] \tag{225}$$
$$\ell_E \equiv i[32..63] \tag{226}$$
$$\ell_M \equiv i[64..95] \tag{227}$$
$$B \equiv i[96..(95+\ell_B)] \tag{228}$$
$$E \equiv i[(96+\ell_B)..(95+\ell_B+\ell_E)] \tag{229}$$
$$M \equiv i[(96+\ell_B+\ell_E)..(95+\ell_B+\ell_E+\ell_M)] \tag{230}$$
$$i[x] \equiv \begin{cases} I_{\mathbf{d}}[x] & \text{if } x < |I_{\mathbf{d}}| \\ 0 & \text{otherwise} \end{cases} \tag{231}$$

E.1. **zkSNARK Related Precompiled Contracts.** We choose two numbers, both of which are prime.

$$p \equiv 21888242871839275222246405745257275088696311157297823662689037894645226208583 \tag{232}$$
$$q \equiv 21888242871839275222246405745257275088548364400416034343698204186575808495617 \tag{233}$$

Since p is a prime number, $\{0, 1, \ldots, p-1\}$ forms a field with addition and multiplication modulo p. We call this field F_{p}.

We define a set C_1 with

$$C_1 \equiv \{(X, Y) \in F_\mathrm{p} \times F_\mathrm{p} \mid Y^2 = X^3 + 3\} \cup \{(0, 0)\} \tag{234}$$

We define a binary operation $+$ on C_1 for distinct elements $(X_1, Y_1), (X_2, Y_2)$ with

$$\begin{aligned} (X_1, Y_1) + (X_2, Y_2) &\equiv \begin{cases} (X, Y) & \text{if } X_1 \neq X_2 \\ (0, 0) & \text{otherwise} \end{cases} \\ \lambda &\equiv \frac{Y_2 - Y_1}{X_2 - X_1} \\ X &\equiv \lambda^2 - X_1 - X_2 \\ Y &\equiv \lambda(X_1 - X) - Y_1 \end{aligned} \tag{235}$$

In the case where $(X_1, Y_1) = (X_2, Y_2)$, we define $+$ on C_1 with

$$\begin{aligned} (X_1, Y_1) + (X_2, Y_2) &\equiv \begin{cases} (X, Y) & \text{if } Y_1 \neq 0 \\ (0, 0) & \text{otherwise} \end{cases} \\ \lambda &\equiv \frac{3X_1^2}{2Y_1} \\ X &\equiv \lambda^2 - 2X_1 \\ Y &\equiv \lambda(X_1 - X) - Y_1 \end{aligned} \tag{236}$$

$(C_1, +)$ is known to form a group. We define scalar multiplication $\cdot$ with

$$n \cdot P \equiv (0, 0) + \underbrace{P + \cdots + P}_{n} \tag{237}$$

for a natural number n and a point P in C_1.

We define P_1 to be a point $(1, 2)$ on C_1. Let G_1 be the subgroup of $(C_1, +)$ generated by P_1. G_1 is known to be a cyclic group of order q. For a point P in G_1, we define $\log_{P_1}(P)$ to be the smallest natural number n satisfying $n \cdot P_1 = P$. $\log_{P_1}(P)$ is at most $q - 1$.

Let F_{p^2} be a field $F_p[i]/(i^2 + 1)$. We define a set C_2 with

$$C_2 \equiv \{(X, Y) \in F_{p^2} \times F_{p^2} \mid Y^2 = X^3 + 3(i + 9)^{-1}\} \cup \{(0, 0)\} \tag{238}$$

We define a binary operation $+$ and scalar multiplication $\cdot$ with the same equations (235), (236) and (237). $(C_2, +)$ is also known to be a group. We define P_2 in C_2 with

$$\begin{aligned} P_2 \equiv\ & (11559732032986387107991004021392285783925812861821192530917403151452391805634 \times i \\ & +10857046999023057135944570762232829481370756359578518086990519993285655852781, \\ & 4082367875863433681332203403145435568316851327593401208105741076214120093531 \times i \\ & +8495653923123431417604973247489272438418190587263600148770280649306958101930) \end{aligned} \tag{239}$$

We define G_2 to be the subgroup of $(C_2, +)$ generated by P_2. G_2 is known to be the only cyclic group of order q on C_2. For a point P in G_2, we define $\log_{P_2}(P)$ be the smallest natural number n satisfying $n \cdot P_2 = P$. With this definition, $\log_{P_2}(P)$ is at most $q - 1$.

Let G_T be the multiplicative abelian group underlying $F_{q^{12}}$. It is known that a non-degenerate bilinear map $e : G_1 \times G_2 \to G_T$ exists. This bilinear map is a type three pairing. There are several such bilinear maps, it does not matter which is chosen to be e. Let $P_T = e(P_1, P_2)$, a be a set of k points in G_1, and b be a set of k points in G_2. It follows from the definition of a pairing that the following are equivalent

$$\log_{P_1}(a_1) \times \log_{P_2}(b_1) + \cdots + \log_{P_1}(a_k) \times \log_{P_2}(b_k) \equiv 1 \mod q \tag{240}$$

$$\prod_{i=0}^{k} e(a_i, b_i) = P_T \tag{241}$$

Thus the pairing operation provides a method to verify (240).

A 32 byte number $\mathbf{x} \in \mathbf{P}_{256}$ might and might not represent an element of F_p.

$$\delta_\mathrm{p}(\mathbf{x}) \equiv \begin{cases} \mathbf{x} & \text{if } \mathbf{x} < p \\ \varnothing & \text{otherwise} \end{cases} \tag{242}$$

A 64 byte data $\mathbf{x} \in \mathbf{B}_{512}$ might and might not represent an element of G_1.

$$\delta_1(\mathbf{x}) \equiv \begin{cases} g_1 & \text{if } g_1 \in G_1 \\ \varnothing & \text{otherwise} \end{cases} \tag{243}$$

$$g_1 \equiv \begin{cases} (x, y) & \text{if } x \neq \varnothing \wedge y \neq \varnothing \\ \varnothing & \text{otherwise} \end{cases} \tag{244}$$

$$x \equiv \delta_{\mathrm{p}}(\mathbf{x}[0..31]) \tag{245}$$

$$y \equiv \delta_{\mathrm{p}}(\mathbf{x}[32..63]) \tag{246}$$

A 128 byte data $\mathbf{x} \in \mathbf{B}_{1024}$ might and might not represent an element of G_2.

$$\delta_2(\mathbf{x}) \equiv \begin{cases} g_2 & \text{if } g_2 \in G_2 \\ \varnothing & \text{otherwise} \end{cases} \tag{247}$$

$$g_2 \equiv \begin{cases} ((x_0 i + y_0), (x_1 i + y_1)) & \text{if } x_0 \neq \varnothing \wedge y_0 \neq \varnothing \wedge x_1 \neq \varnothing \wedge y_1 \neq \varnothing \\ \varnothing & \text{otherwise} \end{cases} \tag{248}$$

$$x_0 \equiv \delta_{\mathrm{p}}(\mathbf{x}[0..31]) \tag{249}$$

$$y_0 \equiv \delta_{\mathrm{p}}(\mathbf{x}[32..63]) \tag{250}$$

$$x_1 \equiv \delta_{\mathrm{p}}(\mathbf{x}[64..95]) \tag{251}$$

$$y_1 \equiv \delta_{\mathrm{p}}(\mathbf{x}[96..127]) \tag{252}$$

We define $\Xi_{\mathtt{SNARKV}}$ as a precompiled contract which checks if (240) holds, for intended use in zkSNARK verification.

$$\Xi_{\mathtt{SNARKV}} \equiv \Xi_{\mathtt{PRE}} \quad \text{except:} \tag{253}$$

$$\Xi_{\mathtt{SNARKV}}(\boldsymbol{\sigma}, g, I) = \left(\varnothing, 0, A^0, ()\right) \quad \text{if } F \tag{254}$$

$$F \equiv (|I_{\mathbf{d}}| \bmod 192 \neq 0 \vee (\exists j.\ a_{\mathrm{j}} = \varnothing \vee b_{\mathrm{j}} = \varnothing)) \tag{255}$$

$$k = \frac{|I_{\mathbf{d}}|}{192} \tag{256}$$

$$g_{\mathrm{r}} = 60000k + 40000 \tag{257}$$

$$\mathbf{o}[0..31] \equiv \begin{cases} 0x0000000000000000000000000000000000000000000000000000000000000001 & \text{if } v \wedge \neg F \\ 0x0000000000000000000000000000000000000000000000000000000000000000 & \text{if } \neg v \wedge \neg F \end{cases} \tag{258}$$

$$v \equiv (\log_{P_1}(a_1) \times \log_{P_2}(b_1) + \cdots + \log_{P_1}(a_k) \times \log_{P_2}(b_k) \equiv 1 \mod q) \tag{259}$$

$$a_1 \equiv \delta_1(I_{\mathbf{d}}[0..63]) \tag{260}$$

$$b_1 \equiv \delta_2(I_{\mathbf{d}}[64..191]) \tag{261}$$

$$\vdots$$

$$a_k \equiv \delta_1(I_{\mathbf{d}}[(|I_{\mathbf{d}}| - 192)..(|I_{\mathbf{d}}| - 129)]) \tag{262}$$

$$b_k \equiv \delta_2(I_{\mathbf{d}}[(|I_{\mathbf{d}}| - 128)..(|I_{\mathbf{d}}| - 1)]) \tag{263}$$

We define a precompiled contract for addition on G_1.

$$\Xi_{\mathtt{BN_ADD}} \equiv \Xi_{\mathtt{BN_PRE}} \quad \text{except:} \tag{264}$$

$$\Xi_{\mathtt{BN_ADD}}(\boldsymbol{\sigma}, g, I) = \left(\varnothing, 0, A^0, ()\right) \quad \text{if } x = \varnothing \vee y = \varnothing \tag{265}$$

$$g_{\mathrm{r}} = 500 \tag{266}$$

$$\mathbf{o} \equiv \delta_1^{-1}(x + y) \quad \text{where } + \text{ is the group operation in } G_1 \tag{267}$$

$$x \equiv \delta_1\left(\bar{I}_{\mathbf{d}}[0..63]\right) \tag{268}$$

$$y \equiv \delta_1\left(\bar{I}_{\mathbf{d}}[64..127]\right) \tag{269}$$

$$\bar{I}_{\mathbf{d}}[x] \equiv \begin{cases} I_{\mathbf{d}}[x] & \text{if } x < |I_{\mathbf{d}}| \\ 0 & \text{otherwise} \end{cases} \tag{270}$$

We define a precompiled contract for scalar multiplication on G_1, where $\bar{I}_{\mathbf{d}}$ is defined in (270).

$$\Xi_{\mathtt{BN_MUL}} \equiv \Xi_{\mathtt{PRE}} \quad \text{except:} \tag{271}$$

$$\Xi_{\mathtt{BN_MUL}}(\boldsymbol{\sigma}, g, I) = \left(\varnothing, 0, A^0, ()\right) \quad \text{if } x = \varnothing \tag{272}$$

$$g_{\mathrm{r}} = 40000 \tag{273}$$

$$\mathbf{o} \equiv \delta_1^{-1}(n \cdot x) \quad \text{where } \cdot \text{ is the scalar multiplication in } G_1 \tag{274}$$

$$n \equiv \bar{I}_{\mathbf{d}}[0..31] \tag{275}$$

$$x \equiv \delta_1\left(\bar{I}_{\mathbf{d}}[32..95]\right) \tag{276}$$

Appendix F. Signing Transactions

The method of signing transactions is similar to the 'Electrum style signatures' as defined by Arnaud et al. [2017], heading "Managing styles with Radium" in the bullet point list. This method utilises the SECP-256k1 curve as described by Courtois et al. [2014], and is implemented similarly to as described by Gura et al. [2004] on p. 9 of 15, para. 3.

It is assumed that the sender has a valid private key p_{r}, which is a randomly selected positive integer (represented as a byte array of length 32 in big-endian form) in the range $[1, \mathtt{secp256k1n} - 1]$.

We assume the existence of functions ECDSAPUBKEY, ECDSASIGN and ECDSARECOVER. These are formally defined in the literature, e.g. by Johnson et al. [2001].

$$\mathtt{ECDSAPUBKEY}(p_{\mathrm{r}} \in \mathbb{B}_{32}) \equiv p_{\mathrm{u}} \in \mathbb{B}_{64} \tag{277}$$

$$\mathtt{ECDSASIGN}(e \in \mathbb{B}_{32}, p_{\mathrm{r}} \in \mathbb{B}_{32}) \equiv (v \in \mathbb{B}_1, r \in \mathbb{B}_{32}, s \in \mathbb{B}_{32}) \tag{278}$$

$$\mathtt{ECDSARECOVER}(e \in \mathbb{B}_{32}, v \in \mathbb{B}_1, r \in \mathbb{B}_{32}, s \in \mathbb{B}_{32}) \equiv p_{\mathrm{u}} \in \mathbb{B}_{64} \tag{279}$$

Where p_{u} is the public key, assumed to be a byte array of size 64 (formed from the concatenation of two positive integers each $< 2^{256}$), p_{r} is the private key, a byte array of size 32 (or a single positive integer in the aforementioned range) and e is the hash of the transaction, $h(T)$. It is assumed that v is the 'recovery identifier'. The recovery identifier is a 1 byte value specifying the parity and finiteness of the coordinates of the curve point for which r is the x-value; this value is in the range of $[27, 30]$, however we declare the upper two possibilities, representing infinite values, invalid. The value 27 represents an even y value and 28 represents an odd y value.

We declare that an ECDSA signature is invalid unless all the following conditions are true[5]:

$$0 < r < \mathtt{secp256k1n} \tag{280}$$

$$0 < s < \mathtt{secp256k1n} \div 2 + 1 \tag{281}$$

$$v \in \{27, 28\} \tag{282}$$

where:

$$\mathtt{secp256k1n} = 115792089237316195423570985008687907852837564279074904382605163141518161494337 \tag{283}$$

For a given private key, p_{r}, the Ethereum address $A(p_{\mathrm{r}})$ (a 160-bit value) to which it corresponds is defined as the right most 160-bits of the Keccak hash of the corresponding ECDSA public key:

$$A(p_{\mathrm{r}}) = \mathcal{B}_{96..255}\big(\mathtt{KEC}\big(\mathtt{ECDSAPUBKEY}(p_{\mathrm{r}})\big)\big) \tag{284}$$

The message hash, $h(T)$, to be signed is the Keccak hash of the transaction. Two different flavors of signing schemes are available. One operates without the latter three signature components, formally described as T_{r}, T_{s} and T_{w}. The other operates on nine elements:

$$L_S(T) \equiv \begin{cases} (T_{\mathrm{n}}, T_{\mathrm{p}}, T_{\mathrm{g}}, T_{\mathrm{t}}, T_{\mathrm{v}}, \mathbf{p}) & \text{if } v \in \{27, 28\} \\ (T_{\mathrm{n}}, T_{\mathrm{p}}, T_{\mathrm{g}}, T_{\mathrm{t}}, T_{\mathrm{v}}, \mathbf{p}, \mathtt{chain_id}, (), ()) & \text{otherwise} \end{cases} \tag{285}$$

where

$$\mathbf{p} \equiv \begin{cases} T_{\mathbf{i}} & \text{if } T_{\mathrm{t}} = 0 \\ T_{\mathbf{d}} & \text{otherwise} \end{cases}$$

$$h(T) \equiv \mathtt{KEC}(L_S(T)) \tag{286}$$

The signed transaction $G(T, p_{\mathrm{r}})$ is defined as:

$$G(T, p_{\mathrm{r}}) \equiv T \quad \text{except:} \tag{287}$$

$$(T_{\mathrm{w}}, T_{\mathrm{r}}, T_{\mathrm{s}}) = \mathtt{ECDSASIGN}(h(T), p_{\mathrm{r}}) \tag{288}$$

Reiterating from previously:

$$T_{\mathrm{r}} = r \tag{289}$$

$$T_{\mathrm{s}} = s \tag{290}$$

T_{w} is either the recovery identifier or 'chain identifier doubled plus 35 or 36'. In the second case, where v is the chain identifier doubled plus 35 or 36, the values 35 and 36 assume the role of the 'recovery identifier' by specifying the parity of y, with the value 35 representing an even value and 36 representing an odd value.

We may then define the sender function S of the transaction as:

$$S(T) \equiv \mathcal{B}_{96..255}\big(\mathtt{KEC}\big(\mathtt{ECDSARECOVER}(h(T), v_0, T_{\mathrm{r}}, T_{\mathrm{s}})\big)\big) \tag{291}$$

$$v_0 \equiv \begin{cases} T_{\mathrm{w}} & \text{if } T_{\mathrm{w}} \in \{27, 28\} \\ 28 - (T_{\mathrm{w}} \bmod 2) & \text{otherwise} \end{cases} \tag{292}$$

The assertion that the sender of a signed transaction equals the address of the signer should be self-evident:

$$\forall T : \forall p_{\mathrm{r}} : S(G(T, p_{\mathrm{r}})) \equiv A(p_{\mathrm{r}}) \tag{293}$$

[5] A signature of a transaction can be valid not only with a recovery identifier but with some other numbers. See how the component T_w of a transaction is interpreted.

Appendix G. Fee Schedule

The fee schedule G is a tuple of 31 scalar values corresponding to the relative costs, in gas, of a number of abstract operations that a transaction may effect.

Name	Value	Description*
G_{zero}	0	Nothing paid for operations of the set W_{zero}.
G_{base}	2	Amount of gas to pay for operations of the set W_{base}.
$G_{verylow}$	3	Amount of gas to pay for operations of the set $W_{verylow}$.
G_{low}	5	Amount of gas to pay for operations of the set W_{low}.
G_{mid}	8	Amount of gas to pay for operations of the set W_{mid}.
G_{high}	10	Amount of gas to pay for operations of the set W_{high}.
$G_{extcode}$	700	Amount of gas to pay for operations of the set $W_{extcode}$.
$G_{balance}$	400	Amount of gas to pay for a BALANCE operation.
G_{sload}	200	Paid for a SLOAD operation.
$G_{jumpdest}$	1	Paid for a JUMPDEST operation.
G_{sset}	20000	Paid for an SSTORE operation when the storage value is set to non-zero from zero.
G_{sreset}	5000	Paid for an SSTORE operation when the storage value's zeroness remains unchanged or is set to zero.
R_{sclear}	15000	Refund given (added into refund counter) when the storage value is set to zero from non-zero.
$R_{selfdestruct}$	24000	Refund given (added into refund counter) for self-destructing an account.
$G_{selfdestruct}$	5000	Amount of gas to pay for a SELFDESTRUCT operation.
G_{create}	32000	Paid for a CREATE operation.
$G_{codedeposit}$	200	Paid per byte for a CREATE operation to succeed in placing code into state.
G_{call}	700	Paid for a CALL operation.
$G_{callvalue}$	9000	Paid for a non-zero value transfer as part of the CALL operation.
$G_{callstipend}$	2300	A stipend for the called contract subtracted from $G_{callvalue}$ for a non-zero value transfer.
$G_{newaccount}$	25000	Paid for a CALL or SELFDESTRUCT operation which creates an account.
G_{exp}	10	Partial payment for an EXP operation.
$G_{expbyte}$	50	Partial payment when multiplied by $\lceil \log_{256}(exponent) \rceil$ for the EXP operation.
G_{memory}	3	Paid for every additional word when expanding memory.
G_{txcreate}	32000	Paid by all contract-creating transactions after the *Homestead* transition.
$G_{txdatazero}$	4	Paid for every zero byte of data or code for a transaction.
$G_{txdatanonzero}$	68	Paid for every non-zero byte of data or code for a transaction.
$G_{transaction}$	21000	Paid for every transaction.
G_{log}	375	Partial payment for a LOG operation.
G_{logdata}	8	Paid for each byte in a LOG operation's data.
G_{logtopic}	375	Paid for each topic of a LOG operation.
G_{sha3}	30	Paid for each SHA3 operation.
$G_{sha3word}$	6	Paid for each word (rounded up) for input data to a SHA3 operation.
G_{copy}	3	Partial payment for *COPY operations, multiplied by words copied, rounded up.
$G_{blockhash}$	20	Payment for BLOCKHASH operation.
$G_{quaddivisor}$	100	The quadratic coefficient of the input sizes of the exponentiation-over-modulo precompiled contract.

Appendix H. Virtual Machine Specification

When interpreting 256-bit binary values as integers, the representation is big-endian.

When a 256-bit machine datum is converted to and from a 160-bit address or hash, the rightwards (low-order for BE) 20 bytes are used and the left most 12 are discarded or filled with zeroes, thus the integer values (when the bytes are interpreted as big-endian) are equivalent.

H.1. **Gas Cost.** The general gas cost function, C, is defined as:

(294)

$$C(\boldsymbol{\sigma}, \boldsymbol{\mu}, I) \equiv C_{mem}(\boldsymbol{\mu}'_{\mathrm{i}}) - C_{mem}(\boldsymbol{\mu}_{\mathrm{i}}) + \begin{cases} C_{\text{SSTORE}}(\boldsymbol{\sigma}, \boldsymbol{\mu}) & \text{if} \quad w = \text{SSTORE} \\ G_{exp} & \text{if} \quad w = \text{EXP} \wedge \boldsymbol{\mu}_{\mathbf{s}}[1] = 0 \\ G_{exp} + G_{expbyte} \times (1 + \lfloor \log_{256}(\boldsymbol{\mu}_{\mathbf{s}}[1]) \rfloor) & \text{if} \quad w = \text{EXP} \wedge \boldsymbol{\mu}_{\mathbf{s}}[1] > 0 \\ G_{verylow} + G_{copy} \times \lceil \boldsymbol{\mu}_{\mathbf{s}}[2] \div 32 \rceil & \text{if} \quad w = \text{CALLDATACOPY} \vee \\ & \quad \text{CODECOPY} \vee \text{RETURNDATACOPY} \\ G_{extcode} + G_{copy} \times \lceil \boldsymbol{\mu}_{\mathbf{s}}[3] \div 32 \rceil & \text{if} \quad w = \text{EXTCODECOPY} \\ G_{log} + G_{logdata} \times \boldsymbol{\mu}_{\mathbf{s}}[1] & \text{if} \quad w = \text{LOG0} \\ G_{log} + G_{logdata} \times \boldsymbol{\mu}_{\mathbf{s}}[1] + G_{logtopic} & \text{if} \quad w = \text{LOG1} \\ G_{log} + G_{logdata} \times \boldsymbol{\mu}_{\mathbf{s}}[1] + 2G_{logtopic} & \text{if} \quad w = \text{LOG2} \\ G_{log} + G_{logdata} \times \boldsymbol{\mu}_{\mathbf{s}}[1] + 3G_{logtopic} & \text{if} \quad w = \text{LOG3} \\ G_{log} + G_{logdata} \times \boldsymbol{\mu}_{\mathbf{s}}[1] + 4G_{logtopic} & \text{if} \quad w = \text{LOG4} \\ C_{\text{CALL}}(\boldsymbol{\sigma}, \boldsymbol{\mu}) & \text{if} \quad w = \text{CALL} \vee \text{CALLCODE} \vee \\ & \quad \text{DELEGATECALL} \\ C_{\text{SELFDESTRUCT}}(\boldsymbol{\sigma}, \boldsymbol{\mu}) & \text{if} \quad w = \text{SELFDESTRUCT} \\ G_{create} & \text{if} \quad w = \text{CREATE} \\ G_{sha3} + G_{sha3word} \lceil \mathbf{s}[1] \div 32 \rceil & \text{if} \quad w = \text{SHA3} \\ G_{jumpdest} & \text{if} \quad w = \text{JUMPDEST} \\ G_{sload} & \text{if} \quad w = \text{SLOAD} \\ G_{zero} & \text{if} \quad w \in W_{zero} \\ G_{base} & \text{if} \quad w \in W_{base} \\ G_{verylow} & \text{if} \quad w \in W_{verylow} \\ G_{\text{low}} & \text{if} \quad w \in W_{\text{low}} \\ G_{mid} & \text{if} \quad w \in W_{mid} \\ G_{\text{high}} & \text{if} \quad w \in W_{\text{high}} \\ G_{extcode} & \text{if} \quad w \in W_{extcode} \\ G_{balance} & \text{if} \quad w = \text{BALANCE} \\ G_{blockhash} & \text{if} \quad w = \text{BLOCKHASH} \end{cases}$$

(295)

$$w \equiv \begin{cases} I_{\mathbf{b}}[\boldsymbol{\mu}_{\mathrm{pc}}] & \text{if} \quad \boldsymbol{\mu}_{\mathrm{pc}} < \|I_{\mathbf{b}}\| \\ \text{STOP} & \text{otherwise} \end{cases}$$

where:

(296)

$$C_{mem}(a) \equiv G_{memory} \cdot a + \left\lfloor \frac{a^2}{512} \right\rfloor$$

with C_{CALL}, $C_{\text{SELFDESTRUCT}}$ and C_{SSTORE} as specified in the appropriate section below. We define the following subsets of instructions:

$W_{zero} = \{$STOP, RETURN, REVERT$\}$
$W_{base} = \{$ADDRESS, ORIGIN, CALLER, CALLVALUE, CALLDATASIZE, CODESIZE, GASPRICE, COINBASE, TIMESTAMP, NUMBER, DIFFICULTY, GASLIMIT, RETURNDATASIZE, POP, PC, MSIZE, GAS$\}$
$W_{verylow} = \{$ADD, SUB, NOT, LT, GT, SLT, SGT, EQ, ISZERO, AND, OR, XOR, BYTE, CALLDATALOAD, MLOAD, MSTORE, MSTORE8, PUSH*, DUP*, SWAP*$\}$
$W_{\text{low}} = \{$MUL, DIV, SDIV, MOD, SMOD, SIGNEXTEND$\}$
$W_{mid} = \{$ADDMOD, MULMOD, JUMP$\}$
$W_{\text{high}} = \{$JUMPI$\}$
$W_{extcode} = \{$EXTCODESIZE$\}$

Note the memory cost component, given as the product of G_{memory} and the maximum of 0 & the ceiling of the number of words in size that the memory must be over the current number of words, $\boldsymbol{\mu}_{\mathrm{i}}$ in order that all accesses reference valid memory whether for read or write. Such accesses must be for non-zero number of bytes.

Referencing a zero length range (e.g. by attempting to pass it as the input range to a CALL) does not require memory to be extended to the beginning of the range. $\boldsymbol{\mu}'_{\mathrm{i}}$ is defined as this new maximum number of words of active memory; special-cases are given where these two are not equal.

Note also that C_{mem} is the memory cost function (the expansion function being the difference between the cost before and after). It is a polynomial, with the higher-order coefficient divided and floored, and thus linear up to 724B of memory used, after which it costs substantially more.

While defining the instruction set, we defined the memory-expansion for range function, M, thus:

$$M(s, f, l) \equiv \begin{cases} s & \text{if} \quad l = 0 \\ \max(s, \lceil (f + l) \div 32 \rceil) & \text{otherwise} \end{cases} \tag{297}$$

Another useful function is "all but one 64th" function L defined as:

$$L(n) \equiv n - \lfloor n/64 \rfloor \tag{298}$$

H.2. **Instruction Set.** As previously specified in section 9, these definitions take place in the final context there. In particular we assume O is the EVM state-progression function and define the terms pertaining to the next cycle's state $(\boldsymbol{\sigma}', \boldsymbol{\mu}')$ such that:

$$O(\boldsymbol{\sigma}, \boldsymbol{\mu}, A, I) \equiv (\boldsymbol{\sigma}', \boldsymbol{\mu}', A', I) \quad \text{with exceptions, as noted} \tag{299}$$

Here given are the various exceptions to the state transition rules given in section 9 specified for each instruction, together with the additional instruction-specific definitions of J and C. For each instruction, also specified is α, the additional items placed on the stack and δ, the items removed from stack, as defined in section 9.

0s: Stop and Arithmetic Operations

All arithmetic is modulo 2^{256} unless otherwise noted. The zero-th power of zero 0^0 is defined to be one.

Value	Mnemonic	δ	α	Description
0x00	STOP	0	0	Halts execution.
0x01	ADD	2	1	Addition operation. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \boldsymbol{\mu}_{\mathbf{s}}[0] + \boldsymbol{\mu}_{\mathbf{s}}[1]$
0x02	MUL	2	1	Multiplication operation. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \boldsymbol{\mu}_{\mathbf{s}}[0] \times \boldsymbol{\mu}_{\mathbf{s}}[1]$
0x03	SUB	2	1	Subtraction operation. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \boldsymbol{\mu}_{\mathbf{s}}[0] - \boldsymbol{\mu}_{\mathbf{s}}[1]$
0x04	DIV	2	1	Integer division operation. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \begin{cases} 0 & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[1] = 0 \\ \lfloor \boldsymbol{\mu}_{\mathbf{s}}[0] \div \boldsymbol{\mu}_{\mathbf{s}}[1] \rfloor & \text{otherwise} \end{cases}$
0x05	SDIV	2	1	Signed integer division operation (truncated). $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \begin{cases} 0 & \text{if } \boldsymbol{\mu}_{\mathbf{s}}[1] = 0 \\ -2^{255} & \text{if } \boldsymbol{\mu}_{\mathbf{s}}[0] = -2^{255} \wedge \boldsymbol{\mu}_{\mathbf{s}}[1] = -1 \\ \mathbf{sgn}(\boldsymbol{\mu}_{\mathbf{s}}[0] \div \boldsymbol{\mu}_{\mathbf{s}}[1]) \lfloor \lvert \boldsymbol{\mu}_{\mathbf{s}}[0] \div \boldsymbol{\mu}_{\mathbf{s}}[1] \rvert \rfloor & \text{otherwise} \end{cases}$ Where all values are treated as two's complement signed 256-bit integers. Note the overflow semantic when -2^{255} is negated.
0x06	MOD	2	1	Modulo remainder operation. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \begin{cases} 0 & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[1] = 0 \\ \boldsymbol{\mu}_{\mathbf{s}}[0] \bmod \boldsymbol{\mu}_{\mathbf{s}}[1] & \text{otherwise} \end{cases}$
0x07	SMOD	2	1	Signed modulo remainder operation. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \begin{cases} 0 & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[1] = 0 \\ \mathbf{sgn}(\boldsymbol{\mu}_{\mathbf{s}}[0])(\lvert \boldsymbol{\mu}_{\mathbf{s}}[0] \rvert \bmod \lvert \boldsymbol{\mu}_{\mathbf{s}}[1] \rvert) & \text{otherwise} \end{cases}$ Where all values are treated as two's complement signed 256-bit integers.
0x08	ADDMOD	3	1	Modulo addition operation. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \begin{cases} 0 & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[2] = 0 \\ (\boldsymbol{\mu}_{\mathbf{s}}[0] + \boldsymbol{\mu}_{\mathbf{s}}[1]) \bmod \boldsymbol{\mu}_{\mathbf{s}}[2] & \text{otherwise} \end{cases}$ All intermediate calculations of this operation are not subject to the 2^{256} modulo.
0x09	MULMOD	3	1	Modulo multiplication operation. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \begin{cases} 0 & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[2] = 0 \\ (\boldsymbol{\mu}_{\mathbf{s}}[0] \times \boldsymbol{\mu}_{\mathbf{s}}[1]) \bmod \boldsymbol{\mu}_{\mathbf{s}}[2] & \text{otherwise} \end{cases}$ All intermediate calculations of this operation are not subject to the 2^{256} modulo.
0x0a	EXP	2	1	Exponential operation. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \boldsymbol{\mu}_{\mathbf{s}}[0]^{\boldsymbol{\mu}_{\mathbf{s}}[1]}$
0x0b	SIGNEXTEND	2	1	Extend length of two's complement signed integer. $\forall i \in [0..255] : \boldsymbol{\mu}'_{\mathbf{s}}[0]_{\mathrm{i}} \equiv \begin{cases} \boldsymbol{\mu}_{\mathbf{s}}[1]_{\mathrm{t}} & \text{if} \quad i \leqslant t \quad \text{where } t = 256 - 8(\boldsymbol{\mu}_{\mathbf{s}}[0] + 1) \\ \boldsymbol{\mu}_{\mathbf{s}}[1]_{\mathrm{i}} & \text{otherwise} \end{cases}$

$\boldsymbol{\mu}_{\mathbf{s}}[x]_{\mathrm{i}}$ gives the ith bit (counting from zero) of $\boldsymbol{\mu}_{\mathbf{s}}[x]$

				10s: Comparison & Bitwise Logic Operations
Value	**Mnemonic**	δ	α	**Description**
0x10	LT	2	1	Less-than comparison. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \begin{cases} 1 & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[0] < \boldsymbol{\mu}_{\mathbf{s}}[1] \\ 0 & \text{otherwise} \end{cases}$
0x11	GT	2	1	Greater-than comparison. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \begin{cases} 1 & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[0] > \boldsymbol{\mu}_{\mathbf{s}}[1] \\ 0 & \text{otherwise} \end{cases}$
0x12	SLT	2	1	Signed less-than comparison. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \begin{cases} 1 & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[0] < \boldsymbol{\mu}_{\mathbf{s}}[1] \\ 0 & \text{otherwise} \end{cases}$ Where all values are treated as two's complement signed 256-bit integers.
0x13	SGT	2	1	Signed greater-than comparison. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \begin{cases} 1 & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[0] > \boldsymbol{\mu}_{\mathbf{s}}[1] \\ 0 & \text{otherwise} \end{cases}$ Where all values are treated as two's complement signed 256-bit integers.
0x14	EQ	2	1	Equality comparison. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \begin{cases} 1 & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[0] = \boldsymbol{\mu}_{\mathbf{s}}[1] \\ 0 & \text{otherwise} \end{cases}$
0x15	ISZERO	1	1	Simple not operator. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \begin{cases} 1 & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[0] = 0 \\ 0 & \text{otherwise} \end{cases}$
0x16	AND	2	1	Bitwise AND operation. $\forall i \in [0..255] : \boldsymbol{\mu}'_{\mathbf{s}}[0]_{\mathrm{i}} \equiv \boldsymbol{\mu}_{\mathbf{s}}[0]_{\mathrm{i}} \wedge \boldsymbol{\mu}_{\mathbf{s}}[1]_{\mathrm{i}}$
0x17	OR	2	1	Bitwise OR operation. $\forall i \in [0..255] : \boldsymbol{\mu}'_{\mathbf{s}}[0]_{\mathrm{i}} \equiv \boldsymbol{\mu}_{\mathbf{s}}[0]_{\mathrm{i}} \vee \boldsymbol{\mu}_{\mathbf{s}}[1]_{\mathrm{i}}$
0x18	XOR	2	1	Bitwise XOR operation. $\forall i \in [0..255] : \boldsymbol{\mu}'_{\mathbf{s}}[0]_{\mathrm{i}} \equiv \boldsymbol{\mu}_{\mathbf{s}}[0]_{\mathrm{i}} \oplus \boldsymbol{\mu}_{\mathbf{s}}[1]_{\mathrm{i}}$
0x19	NOT	1	1	Bitwise NOT operation. $\forall i \in [0..255] : \boldsymbol{\mu}'_{\mathbf{s}}[0]_{\mathrm{i}} \equiv \begin{cases} 1 & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[0]_{\mathrm{i}} = 0 \\ 0 & \text{otherwise} \end{cases}$
0x1a	BYTE	2	1	Retrieve single byte from word. $\forall i \in [0..255] : \boldsymbol{\mu}'_{\mathbf{s}}[0]_{\mathrm{i}} \equiv \begin{cases} \boldsymbol{\mu}_{\mathbf{s}}[1]_{(i+8\boldsymbol{\mu}_{\mathbf{s}}[0])} & \text{if} \quad i < 8 \wedge \boldsymbol{\mu}_{\mathbf{s}}[0] < 32 \\ 0 & \text{otherwise} \end{cases}$ For the Nth byte, we count from the left (i.e. N=0 would be the most significant in big endian).

				20s: SHA3
Value	**Mnemonic**	δ	α	**Description**
0x20	SHA3	2	1	Compute Keccak-256 hash. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \texttt{Keccak}(\boldsymbol{\mu}_{\mathbf{m}}[\boldsymbol{\mu}_{\mathbf{s}}[0] \ldots (\boldsymbol{\mu}_{\mathbf{s}}[0] + \boldsymbol{\mu}_{\mathbf{s}}[1] - 1)])$ $\boldsymbol{\mu}'_{\mathrm{i}} \equiv M(\boldsymbol{\mu}_{\mathrm{i}}, \boldsymbol{\mu}_{\mathbf{s}}[0], \boldsymbol{\mu}_{\mathbf{s}}[1])$

				30s: Environmental Information
Value	**Mnemonic**	δ	α	**Description**
0x30	ADDRESS	0	1	Get address of currently executing account. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv I_{\mathrm{a}}$
0x31	BALANCE	1	1	Get balance of the given account. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \begin{cases} \boldsymbol{\sigma}[\boldsymbol{\mu}_{\mathbf{s}}[0]]_{\mathrm{b}} & \text{if} \quad \boldsymbol{\sigma}[\boldsymbol{\mu}_{\mathbf{s}}[0] \mod 2^{160}] \neq \varnothing \\ 0 & \text{otherwise} \end{cases}$
0x32	ORIGIN	0	1	Get execution origination address. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv I_{\mathrm{o}}$ This is the sender of original transaction; it is never an account with non-empty associated code.
0x33	CALLER	0	1	Get caller address. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv I_{\mathrm{s}}$ This is the address of the account that is directly responsible for this execution.
0x34	CALLVALUE	0	1	Get deposited value by the instruction/transaction responsible for this execution. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv I_{\mathrm{v}}$
0x35	CALLDATALOAD	1	1	Get input data of current environment. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv I_{\mathbf{d}}[\boldsymbol{\mu}_{\mathbf{s}}[0] \dots (\boldsymbol{\mu}_{\mathbf{s}}[0] + 31)] \quad \text{with} \quad I_{\mathbf{d}}[x] = 0 \quad \text{if} \quad x \geqslant \lVert I_{\mathbf{d}} \rVert$ This pertains to the input data passed with the message call instruction or transaction.
0x36	CALLDATASIZE	0	1	Get size of input data in current environment. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \lVert I_{\mathbf{d}} \rVert$ This pertains to the input data passed with the message call instruction or transaction.
0x37	CALLDATACOPY	3	0	Copy input data in current environment to memory. $\forall_{i \in \{0 \dots \boldsymbol{\mu}_{\mathbf{s}}[2]-1\}} \boldsymbol{\mu}'_{\mathbf{m}}[\boldsymbol{\mu}_{\mathbf{s}}[0] + i] \equiv \begin{cases} I_{\mathbf{d}}[\boldsymbol{\mu}_{\mathbf{s}}[1] + i] & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[1] + i < \lVert I_{\mathbf{d}} \rVert \\ 0 & \text{otherwise} \end{cases}$ The additions in $\boldsymbol{\mu}_{\mathbf{s}}[1] + i$ are not subject to the 2^{256} modulo. $\boldsymbol{\mu}'_{\mathrm{i}} \equiv M(\boldsymbol{\mu}_{\mathrm{i}}, \boldsymbol{\mu}_{\mathbf{s}}[0], \boldsymbol{\mu}_{\mathbf{s}}[2])$ This pertains to the input data passed with the message call instruction or transaction.
0x38	CODESIZE	0	1	Get size of code running in current environment. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \lVert I_{\mathbf{b}} \rVert$
0x39	CODECOPY	3	0	Copy code running in current environment to memory. $\forall_{i \in \{0 \dots \boldsymbol{\mu}_{\mathbf{s}}[2]-1\}} \boldsymbol{\mu}'_{\mathbf{m}}[\boldsymbol{\mu}_{\mathbf{s}}[0] + i] \equiv \begin{cases} I_{\mathbf{b}}[\boldsymbol{\mu}_{\mathbf{s}}[1] + i] & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[1] + i < \lVert I_{\mathbf{b}} \rVert \\ \text{STOP} & \text{otherwise} \end{cases}$ $\boldsymbol{\mu}'_{\mathrm{i}} \equiv M(\boldsymbol{\mu}_{\mathrm{i}}, \boldsymbol{\mu}_{\mathbf{s}}[0], \boldsymbol{\mu}_{\mathbf{s}}[2])$ The additions in $\boldsymbol{\mu}_{\mathbf{s}}[1] + i$ are not subject to the 2^{256} modulo.
0x3a	GASPRICE	0	1	Get price of gas in current environment. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv I_{\mathrm{p}}$ This is gas price specified by the originating transaction.
0x3b	EXTCODESIZE	1	1	Get size of an account's code. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \lVert \boldsymbol{\sigma}[\boldsymbol{\mu}_{\mathbf{s}}[0] \mod 2^{160}]_{\mathrm{c}} \rVert$
0x3c	EXTCODECOPY	4	0	Copy an account's code to memory. $\forall_{i \in \{0 \dots \boldsymbol{\mu}_{\mathbf{s}}[3]-1\}} \boldsymbol{\mu}'_{\mathbf{m}}[\boldsymbol{\mu}_{\mathbf{s}}[1] + i] \equiv \begin{cases} \mathbf{c}[\boldsymbol{\mu}_{\mathbf{s}}[2] + i] & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[2] + i < \lVert \mathbf{c} \rVert \\ \text{STOP} & \text{otherwise} \end{cases}$ where $\mathbf{c} \equiv \boldsymbol{\sigma}[\boldsymbol{\mu}_{\mathbf{s}}[0] \mod 2^{160}]_{\mathrm{c}}$ $\boldsymbol{\mu}'_{\mathrm{i}} \equiv M(\boldsymbol{\mu}_{\mathrm{i}}, \boldsymbol{\mu}_{\mathbf{s}}[1], \boldsymbol{\mu}_{\mathbf{s}}[3])$ The additions in $\boldsymbol{\mu}_{\mathbf{s}}[2] + i$ are not subject to the 2^{256} modulo.
0x3d	RETURNDATASIZE	0	1	Get size of output data from the previous call from the current environment. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \lVert \boldsymbol{\mu}_{\mathbf{o}} \rVert$
0x3e	RETURNDATACOPY	3	0	Copy output data from the previous call to memory. $\forall_{i \in \{0 \dots \boldsymbol{\mu}_{\mathbf{s}}[2]-1\}} \boldsymbol{\mu}'_{\mathbf{m}}[\boldsymbol{\mu}_{\mathbf{s}}[0] + i] \equiv \begin{cases} \boldsymbol{\mu}_{\mathbf{o}}[\boldsymbol{\mu}_{\mathbf{s}}[1] + i] & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[1] + i < \lVert \boldsymbol{\mu}_{\mathbf{o}} \rVert \\ 0 & \text{otherwise} \end{cases}$ The additions in $\boldsymbol{\mu}_{\mathbf{s}}[1] + i$ are not subject to the 2^{256} modulo. $\boldsymbol{\mu}'_{\mathrm{i}} \equiv M(\boldsymbol{\mu}_{\mathrm{i}}, \boldsymbol{\mu}_{\mathbf{s}}[0], \boldsymbol{\mu}_{\mathbf{s}}[2])$

				40s: Block Information
Value	**Mnemonic**	δ	α	**Description**
0x40	BLOCKHASH	1	1	Get the hash of one of the 256 most recent complete blocks. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv P(I_{\mathbf{H}_{\mathrm{p}}}, \boldsymbol{\mu}_{\mathbf{s}}[0], 0)$ where P is the hash of a block of a particular number, up to a maximum age. 0 is left on the stack if the looked for block number is greater than the current block number or more than 256 blocks behind the current block. $P(h, n, a) \equiv \begin{cases} 0 & \text{if} \quad n > H_{\mathrm{i}} \vee a = 256 \vee h = 0 \\ h & \text{if} \quad n = H_{\mathrm{i}} \\ P(H_{\mathrm{p}}, n, a+1) & \text{otherwise} \end{cases}$ and we assert the header H can be determined from its hash h unless as its hash is the parent h is zero.
0x41	COINBASE	0	1	Get the block's beneficiary address. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv I_{H\mathrm{c}}$
0x42	TIMESTAMP	0	1	Get the block's timestamp. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv I_{H\mathrm{s}}$
0x43	NUMBER	0	1	Get the block's number. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv I_{H\mathrm{i}}$
0x44	DIFFICULTY	0	1	Get the block's difficulty. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv I_{H\mathrm{d}}$
0x45	GASLIMIT	0	1	Get the block's gas limit. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv I_{H\mathrm{l}}$

50s: Stack, Memory, Storage and Flow Operations				
Value	**Mnemonic**	δ	α	**Description**
0x50	POP	1	0	Remove item from stack.
0x51	MLOAD	1	1	Load word from memory. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \boldsymbol{\mu}_{\mathbf{m}}[\boldsymbol{\mu}_{\mathbf{s}}[0] \ldots (\boldsymbol{\mu}_{\mathbf{s}}[0]+31)]$ $\boldsymbol{\mu}'_{\mathrm{i}} \equiv \max(\boldsymbol{\mu}_{\mathrm{i}}, \lceil(\boldsymbol{\mu}_{\mathbf{s}}[0]+32) \div 32\rceil)$ The addition in the calculation of $\boldsymbol{\mu}'_{\mathrm{i}}$ is not subject to the 2^{256} modulo.
0x52	MSTORE	2	0	Save word to memory. $\boldsymbol{\mu}'_{\mathbf{m}}[\boldsymbol{\mu}_{\mathbf{s}}[0] \ldots (\boldsymbol{\mu}_{\mathbf{s}}[0]+31)] \equiv \boldsymbol{\mu}_{\mathbf{s}}[1]$ $\boldsymbol{\mu}'_{\mathrm{i}} \equiv \max(\boldsymbol{\mu}_{\mathrm{i}}, \lceil(\boldsymbol{\mu}_{\mathbf{s}}[0]+32) \div 32\rceil)$ The addition in the calculation of $\boldsymbol{\mu}'_{\mathrm{i}}$ is not subject to the 2^{256} modulo.
0x53	MSTORE8	2	0	Save byte to memory. $\boldsymbol{\mu}'_{\mathbf{m}}[\boldsymbol{\mu}_{\mathbf{s}}[0]] \equiv (\boldsymbol{\mu}_{\mathbf{s}}[1] \bmod 256)$ $\boldsymbol{\mu}'_{\mathrm{i}} \equiv \max(\boldsymbol{\mu}_{\mathrm{i}}, \lceil(\boldsymbol{\mu}_{\mathbf{s}}[0]+1) \div 32\rceil)$ The addition in the calculation of $\boldsymbol{\mu}'_{\mathrm{i}}$ is not subject to the 2^{256} modulo.
0x54	SLOAD	1	1	Load word from storage. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \boldsymbol{\sigma}[I_{\mathrm{a}}]_{\mathbf{s}}[\boldsymbol{\mu}_{\mathbf{s}}[0]]$
0x55	SSTORE	2	0	Save word to storage. $\boldsymbol{\sigma}'[I_{\mathrm{a}}]_{\mathbf{s}}[\boldsymbol{\mu}_{\mathbf{s}}[0]] \equiv \boldsymbol{\mu}_{\mathbf{s}}[1]$ $C_{\mathrm{SSTORE}}(\boldsymbol{\sigma}, \boldsymbol{\mu}) \equiv \begin{cases} G_{sset} & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[1] \neq 0 \ \wedge \ \boldsymbol{\sigma}[I_{\mathrm{a}}]_{\mathbf{s}}[\boldsymbol{\mu}_{\mathbf{s}}[0]] = 0 \\ G_{sreset} & \text{otherwise} \end{cases}$ $A'_r \equiv A_r + \begin{cases} R_{sclear} & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[1] = 0 \ \wedge \ \boldsymbol{\sigma}[I_{\mathrm{a}}]_{\mathbf{s}}[\boldsymbol{\mu}_{\mathbf{s}}[0]] \neq 0 \\ 0 & \text{otherwise} \end{cases}$
0x56	JUMP	1	0	Alter the program counter. $J_{\mathrm{JUMP}}(\boldsymbol{\mu}) \equiv \boldsymbol{\mu}_{\mathbf{s}}[0]$ This has the effect of writing said value to $\boldsymbol{\mu}_{\mathrm{pc}}$. See section 9.
0x57	JUMPI	2	0	Conditionally alter the program counter. $J_{\mathrm{JUMPI}}(\boldsymbol{\mu}) \equiv \begin{cases} \boldsymbol{\mu}_{\mathbf{s}}[0] & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[1] \neq 0 \\ \boldsymbol{\mu}_{\mathrm{pc}} + 1 & \text{otherwise} \end{cases}$ This has the effect of writing said value to $\boldsymbol{\mu}_{\mathrm{pc}}$. See section 9.
0x58	PC	0	1	Get the value of the program counter *prior* to the increment corresponding to this instruction. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \boldsymbol{\mu}_{\mathrm{pc}}$
0x59	MSIZE	0	1	Get the size of active memory in bytes. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv 32\boldsymbol{\mu}_i$
0x5a	GAS	0	1	Get the amount of available gas, including the corresponding reduction for the cost of this instruction. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \boldsymbol{\mu}_g$
0x5b	JUMPDEST	0	0	Mark a valid destination for jumps. This operation has no effect on machine state during execution.

60s & 70s: Push Operations

Value	Mnemonic	δ	α	Description
0x60	PUSH1	0	1	Place 1 byte item on stack. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv c(\boldsymbol{\mu}_{\mathrm{pc}} + 1)$ where $c(x) \equiv \begin{cases} I_{\mathbf{b}}[x] & \text{if} \quad x < \lVert I_{\mathbf{b}} \rVert \\ 0 & \text{otherwise} \end{cases}$ The bytes are read in line from the program code's bytes array. The function c ensures the bytes default to zero if they extend past the limits. The byte is right-aligned (takes the lowest significant place in big endian).
0x61	PUSH2	0	1	Place 2-byte item on stack. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \boldsymbol{c}\big((\boldsymbol{\mu}_{\mathrm{pc}} + 1) \ldots (\boldsymbol{\mu}_{\mathrm{pc}} + 2)\big)$ with $\boldsymbol{c}(\boldsymbol{x}) \equiv (c(\boldsymbol{x}_0), ..., c(\boldsymbol{x}_{\lVert x \rVert - 1}))$ with c as defined as above. The bytes are right-aligned (takes the lowest significant place in big endian).
⋮	⋮	⋮	⋮	⋮
0x7f	PUSH32	0	1	Place 32-byte (full word) item on stack. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \boldsymbol{c}\big((\boldsymbol{\mu}_{\mathrm{pc}} + 1) \ldots (\boldsymbol{\mu}_{\mathrm{pc}} + 32)\big)$ where $\boldsymbol{c}$ is defined as above. The bytes are right-aligned (takes the lowest significant place in big endian).

80s: Duplication Operations

Value	Mnemonic	δ	α	Description
0x80	DUP1	1	2	Duplicate 1st stack item. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \boldsymbol{\mu}_{\mathbf{s}}[0]$
0x81	DUP2	2	3	Duplicate 2nd stack item. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \boldsymbol{\mu}_{\mathbf{s}}[1]$
⋮	⋮	⋮	⋮	⋮
0x8f	DUP16	16	17	Duplicate 16th stack item. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \boldsymbol{\mu}_{\mathbf{s}}[15]$

90s: Exchange Operations

Value	Mnemonic	δ	α	Description
0x90	SWAP1	2	2	Exchange 1st and 2nd stack items. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \boldsymbol{\mu}_{\mathbf{s}}[1]$ $\boldsymbol{\mu}'_{\mathbf{s}}[1] \equiv \boldsymbol{\mu}_{\mathbf{s}}[0]$
0x91	SWAP2	3	3	Exchange 1st and 3rd stack items. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \boldsymbol{\mu}_{\mathbf{s}}[2]$ $\boldsymbol{\mu}'_{\mathbf{s}}[2] \equiv \boldsymbol{\mu}_{\mathbf{s}}[0]$
⋮	⋮	⋮	⋮	⋮
0x9f	SWAP16	17	17	Exchange 1st and 17th stack items. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \boldsymbol{\mu}_{\mathbf{s}}[16]$ $\boldsymbol{\mu}'_{\mathbf{s}}[16] \equiv \boldsymbol{\mu}_{\mathbf{s}}[0]$

a0s: Logging Operations

For all logging operations, the state change is to append an additional log entry on to the substate's log series:

$$\text{A } 1A'_{\mathbf{l}} \equiv A_{\mathbf{l}} \cdot (I_{\text{a}}, \mathbf{t}, \boldsymbol{\mu}_{\mathbf{m}}[\boldsymbol{\mu}_{\mathbf{s}}[0] \ldots (\boldsymbol{\mu}_{\mathbf{s}}[0] + \boldsymbol{\mu}_{\mathbf{s}}[1] - 1)])$$

and to update the memory consumption counter:

$\boldsymbol{\mu}'_{\text{i}} \equiv M(\boldsymbol{\mu}_{\text{i}}, \boldsymbol{\mu}_{\mathbf{s}}[0], \boldsymbol{\mu}_{\mathbf{s}}[1])$

The entry's topic series, $\mathbf{t}$, differs accordingly:

Value	Mnemonic	δ	α	Description
0xa0	LOG0	2	0	Append log record with no topics. $\mathbf{t} \equiv ()$
0xa1	LOG1	3	0	Append log record with one topic. $\mathbf{t} \equiv (\boldsymbol{\mu}_{\mathbf{s}}[2])$
⋮	⋮	⋮	⋮	⋮
0xa4	LOG4	6	0	Append log record with four topics. $\mathbf{t} \equiv (\boldsymbol{\mu}_{\mathbf{s}}[2], \boldsymbol{\mu}_{\mathbf{s}}[3], \boldsymbol{\mu}_{\mathbf{s}}[4], \boldsymbol{\mu}_{\mathbf{s}}[5])$

f0s: System operations

Value	Mnemonic	δ	α	Description
0xf0	CREATE	3	1	Create a new account with associated code. $\mathbf{i} \equiv \boldsymbol{\mu}_{\mathbf{m}}[\boldsymbol{\mu}_{\mathbf{s}}[1] \ldots (\boldsymbol{\mu}_{\mathbf{s}}[1] + \boldsymbol{\mu}_{\mathbf{s}}[2] - 1)]$ $(\boldsymbol{\sigma}', \boldsymbol{\mu}'_{\mathrm{g}}, A^+, \mathbf{o}) \equiv \begin{cases} \Lambda(\boldsymbol{\sigma}^*, I_{\mathrm{a}}, I_{\mathrm{o}}, L(\boldsymbol{\mu}_{\mathrm{g}}), I_{\mathrm{p}}, \boldsymbol{\mu}_{\mathbf{s}}[0], \mathbf{i}, I_{\mathrm{e}} + 1, I_{\mathrm{w}}) & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[0] \leqslant \boldsymbol{\sigma}[I_{\mathrm{a}}]_{\mathrm{b}} \\ & \wedge\, I_{\mathrm{e}} < 1024 \\ (\boldsymbol{\sigma}, \boldsymbol{\mu}_{\mathrm{g}}, \varnothing) & \text{otherwise} \end{cases}$ $\boldsymbol{\sigma}^* \equiv \boldsymbol{\sigma} \quad \text{except} \quad \boldsymbol{\sigma}^*[I_{\mathrm{a}}]_{\mathrm{n}} = \boldsymbol{\sigma}[I_{\mathrm{a}}]_{\mathrm{n}} + 1$ $A' \equiv A \uplus A^+$ which abbreviates: $A'_{\mathbf{s}} \equiv A_{\mathbf{s}} \cup A^+_{\mathbf{s}} \quad \wedge \quad A'_{\mathbf{l}} \equiv A_{\mathbf{l}} \cdot A^+_{\mathbf{l}} \quad \wedge$ $A'_{\mathbf{t}} \equiv A_{\mathbf{t}} \cup A^+_{\mathbf{t}} \wedge \quad A'_{\mathbf{r}} \equiv A_{\mathbf{r}} + A^+_{\mathbf{r}}$ $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv x$ where $x = 0$ if the code execution for this operation failed due to an exceptional halting (or for a REVERT) $\boldsymbol{\sigma}' = \varnothing$, or $I_{\mathrm{e}} = 1024$ (the maximum call depth limit is reached) or $\boldsymbol{\mu}_{\mathbf{s}}[0] > \boldsymbol{\sigma}[I_{\mathrm{a}}]_{\mathrm{b}}$ (balance of the caller is too low to fulfil the value transfer); and otherwise $x = A(I_{\mathrm{a}}, \boldsymbol{\sigma}[I_{\mathrm{a}}]_{\mathrm{n}})$, the address of the newly created account. $\boldsymbol{\mu}'_{\mathrm{i}} \equiv M(\boldsymbol{\mu}_{\mathrm{i}}, \boldsymbol{\mu}_{\mathbf{s}}[1], \boldsymbol{\mu}_{\mathbf{s}}[2])$ $\boldsymbol{\mu}'_{\mathbf{o}} \equiv ()$ Thus the operand order is: value, input offset, input size.
0xf1	CALL	7	1	Message-call into an account. $\mathbf{i} \equiv \boldsymbol{\mu}_{\mathbf{m}}[\boldsymbol{\mu}_{\mathbf{s}}[3] \ldots (\boldsymbol{\mu}_{\mathbf{s}}[3] + \boldsymbol{\mu}_{\mathbf{s}}[4] - 1)]$ $(\boldsymbol{\sigma}', g', A^+, \mathbf{o}) \equiv \begin{cases} \Theta(\boldsymbol{\sigma}, I_{\mathrm{a}}, I_{\mathrm{o}}, t, t, C_{\mathrm{CALLGAS}}(\boldsymbol{\mu}), & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[2] \leqslant \boldsymbol{\sigma}[I_{\mathrm{a}}]_{\mathrm{b}} \wedge \\ \quad I_{\mathrm{p}}, \boldsymbol{\mu}_{\mathbf{s}}[2], \boldsymbol{\mu}_{\mathbf{s}}[2], \mathbf{i}, I_{\mathrm{e}} + 1, I_{\mathrm{w}}) & \quad I_{\mathrm{e}} < 1024 \\ (\boldsymbol{\sigma}, g, \varnothing, ()) & \text{otherwise} \end{cases}$ $n \equiv \min(\{\boldsymbol{\mu}_{\mathbf{s}}[6], |\mathbf{o}|\})$ $\boldsymbol{\mu}'_{\mathbf{m}}[\boldsymbol{\mu}_{\mathbf{s}}[5] \ldots (\boldsymbol{\mu}_{\mathbf{s}}[5] + n - 1)] = \mathbf{o}[0 \ldots (n-1)]$ $\boldsymbol{\mu}'_{\mathbf{o}} = \mathbf{o}$ $\boldsymbol{\mu}'_{\mathrm{g}} \equiv \boldsymbol{\mu}_{\mathrm{g}} + g'$ $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv x$ $A' \equiv A \uplus A^+$ $t \equiv \boldsymbol{\mu}_{\mathbf{s}}[1] \mod 2^{160}$ where $x = 0$ if the code execution for this operation failed due to an exceptional halting (or for a REVERT) $\boldsymbol{\sigma}' = \varnothing$ or if $\boldsymbol{\mu}_{\mathbf{s}}[2] > \boldsymbol{\sigma}[I_{\mathrm{a}}]_{\mathrm{b}}$ (not enough funds) or $I_{\mathrm{e}} = 1024$ (call depth limit reached); $x = 1$ otherwise. $\boldsymbol{\mu}'_{\mathrm{i}} \equiv M(M(\boldsymbol{\mu}_{\mathrm{i}}, \boldsymbol{\mu}_{\mathbf{s}}[3], \boldsymbol{\mu}_{\mathbf{s}}[4]), \boldsymbol{\mu}_{\mathbf{s}}[5], \boldsymbol{\mu}_{\mathbf{s}}[6])$ Thus the operand order is: gas, to, value, in offset, in size, out offset, out size. $C_{\mathrm{CALL}}(\boldsymbol{\sigma}, \boldsymbol{\mu}) \equiv C_{\mathrm{GASCAP}}(\boldsymbol{\sigma}, \boldsymbol{\mu}) + C_{\mathrm{EXTRA}}(\boldsymbol{\sigma}, \boldsymbol{\mu})$ $C_{\mathrm{CALLGAS}}(\boldsymbol{\sigma}, \boldsymbol{\mu}) \equiv \begin{cases} C_{\mathrm{GASCAP}}(\boldsymbol{\sigma}, \boldsymbol{\mu}) + G_{callstipend} & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[2] \neq 0 \\ C_{\mathrm{GASCAP}}(\boldsymbol{\sigma}, \boldsymbol{\mu}) & \text{otherwise} \end{cases}$ $C_{\mathrm{GASCAP}}(\boldsymbol{\sigma}, \boldsymbol{\mu}) \equiv \begin{cases} \min\{L(\boldsymbol{\mu}_{\mathrm{g}} - C_{\mathrm{EXTRA}}(\boldsymbol{\sigma}, \boldsymbol{\mu})), \boldsymbol{\mu}_{\mathbf{s}}[0]\} & \text{if} \quad \boldsymbol{\mu}_{\mathrm{g}} \geq C_{\mathrm{EXTRA}}(\boldsymbol{\sigma}, \boldsymbol{\mu}) \\ \boldsymbol{\mu}_{\mathbf{s}}[0] & \text{otherwise} \end{cases}$ $C_{\mathrm{EXTRA}}(\boldsymbol{\sigma}, \boldsymbol{\mu}) \equiv G_{call} + C_{\mathrm{XFER}}(\boldsymbol{\mu}) + C_{\mathrm{NEW}}(\boldsymbol{\sigma}, \boldsymbol{\mu})$ $C_{\mathrm{XFER}}(\boldsymbol{\mu}) \equiv \begin{cases} G_{callvalue} & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[2] \neq 0 \\ 0 & \text{otherwise} \end{cases}$ $C_{\mathrm{NEW}}(\boldsymbol{\sigma}, \boldsymbol{\mu}) \equiv \begin{cases} G_{newaccount} & \text{if} \quad \mathtt{DEAD}(\boldsymbol{\sigma}, \boldsymbol{\mu}_{\mathbf{s}}[1] \mod 2^{160}) \wedge \boldsymbol{\mu}_{\mathbf{s}}[2] \neq 0 \\ 0 & \text{otherwise} \end{cases}$
0xf2	CALLCODE	7	1	Message-call into this account with an alternative account's code. Exactly equivalent to CALL except: $(\boldsymbol{\sigma}', g', A^+, \mathbf{o}) \equiv \begin{cases} \Theta(\boldsymbol{\sigma}^*, I_{\mathrm{a}}, I_{\mathrm{o}}, I_{\mathrm{a}}, t, C_{\mathrm{CALLGAS}}(\boldsymbol{\mu}), & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[2] \leqslant \boldsymbol{\sigma}[I_{\mathrm{a}}]_{\mathrm{b}} \wedge \\ \quad I_{\mathrm{p}}, \boldsymbol{\mu}_{\mathbf{s}}[2], \boldsymbol{\mu}_{\mathbf{s}}[2], \mathbf{i}, I_{\mathrm{e}} + 1, I_{\mathrm{w}}) & \quad I_{\mathrm{e}} < 1024 \\ (\boldsymbol{\sigma}, g, \varnothing, ()) & \text{otherwise} \end{cases}$ Note the change in the fourth parameter to the call Θ from the 2nd stack value $\boldsymbol{\mu}_{\mathbf{s}}[1]$ (as in CALL) to the present address I_{a}. This means that the recipient is in fact the same account as at present, simply that the code is overwritten.
0xf3	RETURN	2	0	Halt execution returning output data. $H_{\mathrm{RETURN}}(\boldsymbol{\mu}) \equiv \boldsymbol{\mu}_{\mathbf{m}}[\boldsymbol{\mu}_{\mathbf{s}}[0] \ldots (\boldsymbol{\mu}_{\mathbf{s}}[0] + \boldsymbol{\mu}_{\mathbf{s}}[1] - 1)]$ This has the effect of halting the execution at this point with output defined. See section 9. $\boldsymbol{\mu}'_{\mathrm{i}} \equiv M(\boldsymbol{\mu}_{\mathrm{i}}, \boldsymbol{\mu}_{\mathbf{s}}[0], \boldsymbol{\mu}_{\mathbf{s}}[1])$

0xf4	DELEGATECALL	6	1	Message-call into this account with an alternative account's code, but persisting the current values for *sender* and *value*. Compared with CALL, DELEGATECALL takes one fewer arguments. The omitted argument is $\boldsymbol{\mu}_{\mathbf{s}}[2]$. As a result, $\boldsymbol{\mu}_{\mathbf{s}}[3]$, $\boldsymbol{\mu}_{\mathbf{s}}[4]$, $\boldsymbol{\mu}_{\mathbf{s}}[5]$ and $\boldsymbol{\mu}_{\mathbf{s}}[6]$ in the definition of CALL should respectively be replaced with $\boldsymbol{\mu}_{\mathbf{s}}[2]$, $\boldsymbol{\mu}_{\mathbf{s}}[3]$, $\boldsymbol{\mu}_{\mathbf{s}}[4]$ and $\boldsymbol{\mu}_{\mathbf{s}}[5]$. Otherwise it is equivalent to CALL except: $(\boldsymbol{\sigma}', g', A^+, \mathbf{o}) \equiv \begin{cases} \Theta(\boldsymbol{\sigma}^*, I_{\mathrm{s}}, I_{\mathrm{o}}, I_{\mathrm{a}}, t, & \text{if} \quad I_{\mathrm{v}} \leqslant \boldsymbol{\sigma}[I_{\mathrm{a}}]_{\mathrm{b}} \wedge \\ \quad \boldsymbol{\mu}_{\mathbf{s}}[0], I_{\mathrm{p}}, 0, I_{\mathrm{v}}, \mathbf{i}, I_{\mathrm{e}}+1, I_{\mathrm{w}}) & \\ I_{\mathrm{e}} < 1024 & \\ (\boldsymbol{\sigma}, g, \varnothing, ()) & \text{otherwise} \end{cases}$ Note the changes (in addition to that of the fourth parameter) to the second and ninth parameters to the call Θ. This means that the recipient is in fact the same account as at present, simply that the code is overwritten *and* the context is almost entirely identical.
0xfa	STATICCALL	6	1	Static message-call into an account. Exactly equivalent to CALL except: The argument $\boldsymbol{\mu}_{\mathbf{s}}[2]$ is replaced with 0. The deeper argument $\boldsymbol{\mu}_{\mathbf{s}}[3]$, $\boldsymbol{\mu}_{\mathbf{s}}[4]$, $\boldsymbol{\mu}_{\mathbf{s}}[5]$ and $\boldsymbol{\mu}_{\mathbf{s}}[6]$ are respectively replaced with $\boldsymbol{\mu}_{\mathbf{s}}[2]$, $\boldsymbol{\mu}_{\mathbf{s}}[3]$, $\boldsymbol{\mu}_{\mathbf{s}}[4]$ and $\boldsymbol{\mu}_{\mathbf{s}}[5]$. The last argument of Θ is $\perp$.
0xfd	REVERT	2	0	Halt execution reverting state changes but returning data and remaining gas. The effect of this operation is described in (131). For the gas calculation, we use the memory expansion function, $\boldsymbol{\mu}'_{\mathrm{i}} \equiv M(\boldsymbol{\mu}_{\mathrm{i}}, \boldsymbol{\mu}_{\mathbf{s}}[0], \boldsymbol{\mu}_{\mathbf{s}}[1])$
0xfe	INVALID	$\varnothing$	$\varnothing$	Designated invalid instruction.
0xff	SELFDESTRUCT	1	0	Halt execution and register account for later deletion. $A'_{\mathbf{s}} \equiv A_{\mathbf{s}} \cup \{I_{\mathrm{a}}\}$ $\boldsymbol{\sigma}'[r] \equiv \begin{cases} \varnothing & \text{if } \boldsymbol{\sigma}[r] = \varnothing \ \wedge \ \boldsymbol{\sigma}[I_{\mathrm{a}}]_{\mathrm{b}} = 0 \\ (\boldsymbol{\sigma}[r]_{\mathrm{n}}, \boldsymbol{\sigma}[r]_{\mathrm{b}} + \boldsymbol{\sigma}[I_{\mathrm{a}}]_{\mathrm{b}}, \boldsymbol{\sigma}[r]_{\mathbf{s}}, \boldsymbol{\sigma}[r]_{\mathrm{c}}) & \text{if } r \neq I_{\mathrm{a}} \\ (\boldsymbol{\sigma}[r]_{\mathrm{n}}, 0, \boldsymbol{\sigma}[r]_{\mathbf{s}}, \boldsymbol{\sigma}[r]_{\mathrm{c}}) & \text{otherwise} \end{cases}$ where $r = \boldsymbol{\mu}_{\mathbf{s}}[0] \bmod 2^{160}$ $\boldsymbol{\sigma}'[I_{\mathrm{a}}]_{\mathrm{b}} = 0$ $C_{\text{SELFDESTRUCT}}(\boldsymbol{\sigma}, \boldsymbol{\mu}) \equiv G_{selfdestruct} + \begin{cases} G_{newaccount} & \text{if} \quad n \\ 0 & \text{otherwise} \end{cases}$ $n \equiv \texttt{DEAD}(\boldsymbol{\sigma}, \boldsymbol{\mu}_{\mathbf{s}}[0] \mod 2^{160}) \wedge \boldsymbol{\sigma}[I_{\mathrm{a}}]_{\mathrm{b}} \neq 0$

Appendix I. Genesis Block

The genesis block is 15 items, and is specified thus:

$$\big((0_{256}, \texttt{KEC}\big(\texttt{RLP}\big(()\big)\big), 0_{160}, \mathit{stateRoot}, 0, 0, 0_{2048}, 2^{17}, 0, 0, 3141592, \mathit{time}, 0, 0_{256}, \texttt{KEC}\big((42)\big)\big), (), ()\big) \tag{300}$$

Where 0_{256} refers to the parent hash, a 256-bit hash which is all zeroes; 0_{160} refers to the beneficiary address, a 160-bit hash which is all zeroes; 0_{2048} refers to the log bloom, 2048-bit of all zeros; 2^{17} refers to the difficulty; the transaction trie root, receipt trie root, gas used, block number and extradata are both 0, being equivalent to the empty byte array. The sequences of both ommers and transactions are empty and represented by $()$. $\texttt{KEC}\big((42)\big)$ refers to the Keccak hash of a byte array of length one whose first and only byte is of value 42, used for the nonce. $\texttt{KEC}\big(\texttt{RLP}\big(()\big)\big)$ value refers to the hash of the ommer list in RLP, both empty lists.

The proof-of-concept series include a development premine, making the state root hash some value *stateRoot*. Also *time* will be set to the initial timestamp of the genesis block. The latest documentation should be consulted for those values.

Appendix J. Ethash

J.1. **Definitions.** We employ the following definitions:

Name	Value	Description
$J_{wordbytes}$	4	Bytes in word.
$J_{datasetinit}$	2^{30}	Bytes in dataset at genesis.
$J_{datasetgrowth}$	2^{23}	Dataset growth per epoch.
$J_{cacheinit}$	2^{24}	Bytes in cache at genesis.
$J_{cachegrowth}$	2^{17}	Cache growth per epoch.
J_{epoch}	30000	Blocks per epoch.
$J_{mixbytes}$	128	mix length in bytes.
$J_{\text{hashbytes}}$	64	Hash length in bytes.
J_{parents}	256	Number of parents of each dataset element.
$J_{cacherounds}$	3	Number of rounds in cache production.
J_{accesses}	64	Number of accesses in hashimoto loop.

J.2. **Size of dataset and cache.** The size for Ethash's cache $\mathbf{c} \in \mathbb{B}$ and dataset $\mathbf{d} \in \mathbb{B}$ depend on the epoch, which in turn depends on the block number.

$$E_{epoch}(H_{\text{i}}) = \left\lfloor \frac{H_{\text{i}}}{J_{epoch}} \right\rfloor \tag{301}$$

The size of the dataset growth by $J_{datasetgrowth}$ bytes, and the size of the cache by $J_{cachegrowth}$ bytes, every epoch. In order to avoid regularity leading to cyclic behavior, the size must be a prime number. Therefore the size is reduced by a multiple of $J_{mixbytes}$, for the dataset, and $J_{\text{hashbytes}}$ for the cache. Let $d_{size} = \|\mathbf{d}\|$ be the size of the dataset. Which is calculated using

$$d_{size} = E_{\text{prime}}(J_{datasetinit} + J_{datasetgrowth} \cdot E_{epoch} - J_{mixbytes}, J_{mixbytes}) \tag{302}$$

The size of the cache, c_{size}, is calculated using

$$c_{size} = E_{\text{prime}}(J_{cacheinit} + J_{cachegrowth} \cdot E_{epoch} - J_{\text{hashbytes}}, J_{\text{hashbytes}}) \tag{303}$$

$$E_{\text{prime}}(x, y) = \begin{cases} x & \text{if} \quad x/y \in \mathbb{N} \\ E_{\text{prime}}(x - 2 \cdot y, y) & \text{otherwise} \end{cases} \tag{304}$$

J.3. **Dataset generation.** In order to generate the dataset we need the cache $\mathbf{c}$, which is an array of bytes. It depends on the cache size c_{size} and the seed hash $\mathbf{s} \in \mathbb{B}_{32}$.

J.3.1. *Seed hash.* The seed hash is different for every epoch. For the first epoch it is the Keccak-256 hash of a series of 32 bytes of zeros. For every other epoch it is always the Keccak-256 hash of the previous seed hash:

$$\mathbf{s} = C_{seedhash}(H_{\text{i}}) \tag{305}$$

$$C_{seedhash}(H_{\text{i}}) = \begin{cases} \mathbf{0}_{32} & \text{if} \quad E_{epoch}(H_{\text{i}}) = 0 \\ \texttt{KEC}(C_{seedhash}(H_{\text{i}} - J_{epoch})) & \text{otherwise} \end{cases} \tag{306}$$

With $\mathbf{0}_{32}$ being 32 bytes of zeros.

J.3.2. *Cache.* The cache production process involves using the seed hash to first sequentially filling up c_{size} bytes of memory, then performing $J_{cacherounds}$ passes of the RandMemoHash algorithm created by Lerner [2014]. The initial cache $\mathbf{c}'$, being an array of arrays of single bytes, will be constructed as follows.

We define the array $\mathbf{c}_i$, consisting of 64 single bytes, as the ith element of the initial cache:

$$\mathbf{c}_i = \begin{cases} \texttt{KEC512}(\mathbf{s}) & \text{if} \quad i = 0 \\ \texttt{KEC512}(\mathbf{c}_{i-1}) & \text{otherwise} \end{cases} \tag{307}$$

Therefore $\mathbf{c}'$ can be defined as

$$\mathbf{c}'[i] = \mathbf{c}_i \quad \forall \quad i < n \tag{308}$$

$$n = \left\lfloor \frac{c_{size}}{J_{\text{hashbytes}}} \right\rfloor \tag{309}$$

The cache is calculated by performing $J_{cacherounds}$ rounds of the RandMemoHash algorithm to the initial cache $\mathbf{c}'$:

$$\mathbf{c} = E_{cacherounds}(\mathbf{c}', J_{cacherounds}) \tag{310}$$

$$E_{cacherounds}(\mathbf{x}, y) = \begin{cases} \mathbf{x} & \text{if} \quad y = 0 \\ E_{\text{RMH}}(\mathbf{x}) & \text{if} \quad y = 1 \\ E_{cacherounds}(E_{\text{RMH}}(\mathbf{x}), y - 1) & \text{otherwise} \end{cases} \tag{311}$$

Where a single round modifies each subset of the cache as follows:

$$E_{\text{RMH}}(\mathbf{x}) = \big(E_{rmh}(\mathbf{x}, 0), E_{rmh}(\mathbf{x}, 1), ..., E_{rmh}(\mathbf{x}, n - 1)\big) \tag{312}$$

$$E_{rmh}(\mathbf{x}, i) = \texttt{KEC512}(\mathbf{x}'[(i-1+n) \mod n] \oplus \mathbf{x}'[\mathbf{x}'[i][0] \mod n]) \quad \text{with} \quad \mathbf{x}' = \mathbf{x} \quad \text{except} \quad \mathbf{x}'[j] = E_{rmh}(\mathbf{x}, j) \quad \forall \quad j < i \tag{313}$$

J.3.3. *Full dataset calculation.* Essentially, we combine data from J_{parents} pseudorandomly selected cache nodes, and hash that to compute the dataset. The entire dataset is then generated by a number of items, each $J_{\text{hashbytes}}$ bytes in size:

$$\mathbf{d}[i] = E_{datasetitem}(\mathbf{c}, i) \quad \forall \quad i < \left\lfloor \frac{d_{size}}{J_{\text{hashbytes}}} \right\rfloor \tag{314}$$

In order to calculate the single item we use an algorithm inspired by the FNV hash (Glenn Fowler [1991]) in some cases as a non-associative substitute for XOR.

$$E_{\text{FNV}}(\mathbf{x}, \mathbf{y}) = (\mathbf{x} \cdot (\text{0x01000193} \oplus \mathbf{y})) \mod 2^{32} \tag{315}$$

The single item of the dataset can now be calculated as:

$$E_{datasetitem}(\mathbf{c}, i) = E_{\text{parents}}(\mathbf{c}, i, -1, \varnothing) \tag{316}$$

$$E_{\text{parents}}(\mathbf{c}, i, p, \mathbf{m}) = \begin{cases} E_{\text{parents}}(\mathbf{c}, i, p+1, E_{mix}(\mathbf{m}, \mathbf{c}, i, p+1)) & \text{if} \quad p < J_{\text{parents}} - 2 \\ E_{mix}(\mathbf{m}, \mathbf{c}, i, p+1) & \text{otherwise} \end{cases} \tag{317}$$

$$E_{mix}(\mathbf{m}, \mathbf{c}, i, p) = \begin{cases} \texttt{KEC512}(\mathbf{c}[i \mod c_{size}] \oplus i) & \text{if} \quad p = 0 \\ E_{\text{FNV}}\big(\mathbf{m}, \mathbf{c}[E_{\text{FNV}}(i \oplus p, \mathbf{m}[p \mod \lfloor J_{\text{hashbytes}}/J_{wordbytes} \rfloor]) \mod c_{size}]\big) & \text{otherwise} \end{cases} \tag{318}$$

J.4. **Proof-of-work function.** Essentially, we maintain a "mix" $J_{mixbytes}$ bytes wide, and repeatedly sequentially fetch $J_{mixbytes}$ bytes from the full dataset and use the E_{FNV} function to combine it with the mix. $J_{mixbytes}$ bytes of sequential access are used so that each round of the algorithm always fetches a full page from RAM, minimizing translation lookaside buffer misses which ASICs would theoretically be able to avoid.

If the output of this algorithm is below the desired target, then the nonce is valid. Note that the extra application of KEC at the end ensures that there exists an intermediate nonce which can be provided to prove that at least a small amount of work was done; this quick outer PoW verification can be used for anti-DDoS purposes. It also serves to provide statistical assurance that the result is an unbiased, 256 bit number.

The PoW-function returns an array with the compressed mix as its first item and the Keccak-256 hash of the concatenation of the compressed mix with the seed hash as the second item:

$$\texttt{PoW}(H_{\not{H}}, H_{\text{n}}, \mathbf{d}) = \{\mathbf{m}_{\text{c}}(\texttt{KEC}(\texttt{RLP}(L_H(H_{\not{H}}))), H_{\text{n}}, \mathbf{d}), \texttt{KEC}(\mathbf{s}_{\text{h}}(\texttt{KEC}(\texttt{RLP}(L_H(H_{\not{H}}))), H_{\text{n}}) + \mathbf{m}_{\text{c}}(\texttt{KEC}(\texttt{RLP}(L_H(H_{\not{H}}))), H_{\text{n}}, \mathbf{d}))\} \tag{319}$$

With $H_{\not{H}}$ being the hash of the header without the nonce. The compressed mix $\mathbf{m}_{\text{c}}$ is obtained as follows:

$$\mathbf{m}_{\text{c}}(\mathbf{h}, \mathbf{n}, \mathbf{d}) = E_{compress}(E_{\text{accesses}}(\mathbf{d}, \sum_{i=0}^{n_{mix}} \mathbf{s}_{\text{h}}(\mathbf{h}, \mathbf{n}), \mathbf{s}_{\text{h}}(\mathbf{h}, \mathbf{n}), -1), -4) \tag{320}$$

The seed hash being:

$$\mathbf{s}_{\text{h}}(\mathbf{h}, \mathbf{n}) = \texttt{KEC512}(\mathbf{h} + E_{revert}(\mathbf{n})) \tag{321}$$

$E_{revert}(\mathbf{n})$ returns the reverted bytes sequence of the nonce $\mathbf{n}$:

$$E_{revert}(\mathbf{n})[i] = \mathbf{n}[\|\mathbf{n}\| - i] \tag{322}$$

We note that the "+"-operator between two byte sequences results in the concatenation of both sequences.

The dataset $\mathbf{d}$ is obtained as described in section J.3.3.

The number of replicated sequences in the mix is:

$$n_{mix} = \left\lfloor \frac{J_{mixbytes}}{J_{\text{hashbytes}}} \right\rfloor \tag{323}$$

In order to add random dataset nodes to the mix, the E_{accesses} function is used:

$$E_{\text{accesses}}(\mathbf{d}, \mathbf{m}, \mathbf{s}, i) = \begin{cases} E_{mixdataset}(\mathbf{d}, \mathbf{m}, \mathbf{s}, i) & \text{if} \quad i = J_{\text{accesses}} - 2 \\ E_{\text{accesses}}(E_{mixdataset}(\mathbf{d}, \mathbf{m}, \mathbf{s}, i), \mathbf{s}, i+1) & \text{otherwise} \end{cases} \tag{324}$$

$$E_{mixdataset}(\mathbf{d}, \mathbf{m}, \mathbf{s}, i) = E_{\text{FNV}}(\mathbf{m}, E_{newdata}(\mathbf{d}, \mathbf{m}, \mathbf{s}, i) \tag{325}$$

$E_{newdata}$ returns an array with n_{mix} elements:

$$E_{newdata}(\mathbf{d}, \mathbf{m}, \mathbf{s}, i)[j] = \mathbf{d}[E_{\text{FNV}}(i \oplus \mathbf{s}[0], \mathbf{m}[i \mod \left\lfloor \frac{J_{mixbytes}}{J_{wordbytes}} \right\rfloor]) \mod \left\lfloor \frac{d_{size}/J_{\text{hashbytes}}}{n_{mix}} \right\rfloor \cdot n_{mix} + j] \quad \forall \quad j < n_{mix} \tag{326}$$

The mix is compressed as follows:

$$E_{compress}(\mathbf{m}, i) = \begin{cases} \mathbf{m} & \text{if} \quad i \geqslant \|\mathbf{m}\| - 8 \\ E_{compress}(E_{\text{FNV}}(E_{\text{FNV}}(E_{\text{FNV}}(\mathbf{m}[i+4], \mathbf{m}[i+5]), \mathbf{m}[i+6]), \mathbf{m}[i+7]), i+8) & \text{otherwise} \end{cases} \tag{327}$$

Appendix K. Anomalies on the Main Network

K.1. **Deletion of an Account Despite Out-of-gas.** At block 2675119, in the transaction 0xcf416c536ec1a19ed1fb89e4ec7ffb3cf73aa413b3aa9b77d60e4fd81a4296ba, an account at address 0x03 was called and an out-of-gas occurred during the call. Against the equation (195), this added 0x03 in the set of touched addresses, and this transaction turned $\boldsymbol{\sigma}[0x03]$ into $\varnothing$.

Appendix L. List of mathematical symbols

Symbol	Latex Command	Description
$\bigvee$	`\bigvee`	This is the least upper bound, supremum, or join of all elements operated on. Thus it is the greatest element of such elements (Davey and Priestley [2002]).